AL/HR-TR-1996-0028

UNITED STATES AIR FORCE
ARMSTRONG LABORATORY

Accomplishments and Opportunities Report Information Integration for Concurrent Engineering (IICE), Final Report

Richard J. Mayer
Perakath Benjamin
Thomas Blinn
Paula S. deWitte
Madhav Erranguntla
Arthur Keen
Christopher Menzel
Michael K. Painter
Florence Fillion
John W. Crump, IV
Madhavi Lingineni

KNOWLEDGE BASED SYSTEMS, INCORPORATED
One KBSI Place
1500 University Drive East
College Station, Texas 77840-2335

HUMAN RESOURCES DIRECTORATE
LOGISTICS RESEARCH DIVISION
2698 G Street
Wright-Patterson AFB OH 45433-7604

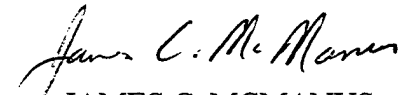September 1997

19971027 016

DTIC QUALITY INSPECTED 3

Human Resources Directorate
Logistics Research Division
2698 G Street
Wright-Patterson AFB OH 45433-7604

NOTICES

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely Government-related procurement, the United States Government incurs no responsibility or any obligation whatsoever. The fact that the Government may have formulated or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication, or otherwise in any manner construed, as licensing the holder, or any other person or corporation, or as conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

The Public Affairs Office has revised this paper and it is releasable to the National Technical Information Service, where it will be available to the general public, including foreign nations.

This paper has been reviewed and is approved for publication.

JAMES C. MCMANUS
Program Manager

BERTRAM W. CREAM, GM-15, DAF
Chief, Logistics Research Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	September 1997	Final - Feb 1991 to Sep 1995

4. TITLE AND SUBTITLE
Accomplishments and Opportunities Report, Information Integration for Concurrent Engineering (IICE)

5. FUNDING NUMBERS
C - F33615-90-C-0012
PE - 63106F
PR - 2940
TA - 01
WU - 08

6. AUTHOR(S)

Richard Mayer	Michael Painter	Paula deWitte	Arther Keen	Joann Sartor
Christopher Menzel	Thomas Blinn	John Crump		MadhaviLingineni
Perakath Benjamin	Florence Fillion	Madhav Erranguntla	James McManus	

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Knowledge Based Systems, Inc.
One KBSI Place
1500 University Drive East
College Station TX 77845

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Armstrong Laboratory
Logistics Research Division
2698 G Street
Wright-Patterson AFB OH 45433 7604

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AL/HR-TP-1996-0028

11. SUPPLEMENTARY NOTES
Armstrong Laboratory Monitor: James C. McManus, AL/HRGA, DSN 785-8049

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*
This document provides an overview and describes the results of the Information Integration for Concurrent Engineering (IICE) project, a four-year program sponsored by the Armstrong Laboratory Logistics Research Division. This program provides the foundations, methods, and tools to effectively implement and evolve toward an information-integrated concurrent engineering environment. In the context of information integration, "information systems" are systems which include both automated and non-automated components responsible for capturing, managing, and controlling information resources. The IICE program investigated information integration concepts supporting this broad definition through eight technical thrusts: Integrated Systems Theory, Three-Schema Architecture, Experimental Tools, Ontology, Frameworks, Methods Engineering, Applications, and Technology Transfer thrusts. In this report, each of these thrusts are discussed in detail, beginning with the goals and objectives of the thrust, the relationship of the thrust to other IICE thrusts, and a summary of the significant accomplishments achieved in the thrust This introduction is then followed by a detailed discussion of the philosophy surrounding each thrust and the technical results derived from each.

14. SUBJECT TERMS

concurrent engineering	information systems	methodology	requirements definition
IDEF	integration	modeling	reengineering
information engineering	knowledge acquisition	method	systems engineering

15. NUMBER OF PAGES
335

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	SAR

Table of Contents

List of Figures

ix

List of Tables

PREFACE

This document is the final report and summarizes the results of the eight technical thrusts (Integrated Systems Theory, Three-Schema Architecture, Experimental Tools, Ontology, Frameworks, Methods Engineering, Applications, and Technology Transfer Thrusts) developed under the Information Integration for Concurrent Engineering (IICE) project, F33615-90-C-0012, funded by Armstrong Laboratory, Logistics Research Division, Wright-Patterson Air Force Base, Ohio 45433, under the technical direction of United States Air Force Captain JoAnn Sartor and Mr. James McManus. The prime contractor for IICE is Knowledge Based Systems, Inc. (KBSI), College Station, Texas. KBSI also acknowledges the technical input to this document made by previous work under the Integrated Information Systems Evolutionary Environment (IISEE) project sponsored by the Armstrong Laboratory Logistics Research Division.

ACCOMPLISHMENTS AND OPPORTUNITIES REPORT

EXECUTIVE SUMMARY

This is the final report for the Information Integration for Concurrent Engineering (IICE) program which is a four-year research and development effort sponsored by the Armstrong Laboratory Logistics Research Division. This program provides the foundations, methods, and tools to effectively implement and evolve toward an information-integrated concurrent engineering environment. Each major component of the effort is structured to promote advancement in the state of the art of information integration practice.

In the context of information integration, "information systems" are systems which include both automated and non-automated components responsible for capturing, managing, and controlling information resources. The IICE program investigated information integration concepts supporting this broad definition. Concurrent engineering was selected as the platform for those investigations..

Summary of Accomplishments

This document summarizes the results of the IICE project, as performed by Knowledge Based Systems, Inc. (KBSI). The project involved eight technical thrusts: Integrated Systems Theory, Three-Schema Architecture, Experimental Tools, Ontology, Frameworks, Methods Engineering, Applications, and Technology Transfer Thrusts. Below is a brief summary of the accomplishments in each IICE thrust.

Table 1. Summary of IICE Accomplishments

Methods Engineering Thrust	• Defined a general approach to method engineering.
	• Produced mature editions of IDEF3, IDEF4, and IDEF5.
	• Developed initial versions of IDEF6, IDEF8, IDEF9, and IDEF14.
	• Developed a standard approach for method formalization.
	• Developed a method meta-language (ISyCL) consistent with the Neutral Information Representation Scheme (NIRS).

Table 1. (Continued)

Experimental Tools Thrust	• Designed and implemented the Level Ø, Level 1, and Level 3 IDSEs.
	• Developed a Framework Processor concept to monitor, manage, and control system development processes[Blinn, Ackley, & Mayer, 1994].
	• Developed the Integrations Services concept to enable and support integration of software applications and utilities.
	• Developed an Evolving System Description, which combines advanced knowledge representation schemes with data consistency management and constraint propagation.
Applications Thrust	• Generated interest in the use and application of the IICE methods.
	• Demonstrated the effectiveness of the IDEF3 and IDEF4 methods to engineer applications.
	• Successfully applied IICE technology in the development of an E-3 PDM Planning System.
Ontology Thrust	• Developed enhanced understanding of the nature of ontology.
	• Developed a characterization of higher-order properties and relations.
	• Assisted in the evolution of the IDEF5 Ontology Capture Method.
	• Defined the role of ontology capture in enterprise integration.
Integrated Systems Theory Thrust	• Developed the Neutral Information Representation Scheme.
	• Developed a detailed model theoretic approach to model integration.
	• Developed a formal, situation theoretic framework to represent dynamic and static information.
	• Developed a formal, situation-based theory of integration and integrated systems.

Table 1. (Concluded)

Frameworks Thrust	• Assisted in the definition of frameworks for various CALS and IDEF organizations. • Developed a framework for object-oriented systems development, implementation, and evolution. • Classified and developed characterizations for various framework types.
Three-Schema Architecture Thrust	• Analyzed various three schema and conceptual schema concepts. • Contributed to the evolution of the Integrated Development Support Environment (IDSE).
Technology Transfer Thrust	• Developed 26 articles documenting IICE results, 6 of which have been published, an additional 7 submitted for publication, and the remaining 13 presented at industry conferences. • Participated in FIPS IDEFØ definition activity with National Institute of Standards and Technology (NIST). • Participated in the Semantic Unification Meta-Model (SUMM) standards activity. • Participated in the IDEF Users Group Interface Definition Language (IDL) standard.

For a more detailed description of these accomplishments, please refer to the sections of this report associated with each thrust.

Organization of the Report

This is the final report for the IICE effort. It has been organized by technical thrusts into the following nine sections:

1.   Executive Summary

2.   Methods Engineering Thrust

3.   Experimental Tools Thrust

4.   Applications Thrust

5.   Ontology Thrust

6. Integrated Systems Theory Thrust

7. Frameworks Thrust

8. Three-Schema Architecture Thrust

9. Technology Transfer Thrust

Each section describes the research activities and results of a particular thrust, beginning with the goals and objectives of the thrust, the relationship of the thrust to other IICE thrusts, and a summary of the significant accomplishments achieved in the thrust. This introduction is followed by a detailed discussion of the philosophy surrounding each thrust and the technical results derived from each.

METHODS ENGINEERING THRUST

This section discusses the overall philosophy, key products, and accomplishments of the Methods Engineering thrust, and provides background information describing the nature, role, and significance of systems engineering methods. Following this introduction, the two main products of the methods engineering thrust are summarized. The first of these products is the foundation for a method engineering discipline to compare, define, extend, and integrate methods. The second product is a family of Integration Definition (IDEF) methods supporting evolution toward an information-integrated enterprise.

Introduction

A key factor limiting the success of enterprise engineering, reengineering, and integration efforts is the lack of effective methods for engaging teams of people in critical life-cycle system development activities. Methods facilitate a scientific approach to problem solving. They guide their practitioners through disciplined, reliable approaches, distilled from expert experience; highlight important objects, relations, and constraints; and hide irrelevant information and unnecessary detail. Methods are designed specifically to raise the performance level (quality and productivity) of the novice practitioner to a level comparable with that of an expert. IDEF methods, a key product of the IICE effort, provide easy-to-use techniques and standard languages of communication that promote good engineering discipline. IDEF methods also improve responsiveness to an environment of rapid and continuous change by helping users to:

- Correctly understand the current environment.

- Propose change.

- Test alternative solutions.

- Predict the impacts of change.

- Implement changes successfully.

As company systems become more model-based, and as the pace and scope of improvement efforts grow, the need for an integrated set of methods becomes increasingly important. The need for an integrated family of methods becomes obvious as organizations seek to minimize duplication while using multiple methods, maintain consistency across models, and ensure that information carried by one model is accurately and efficiently propagated to other models.

Clearly, effective methods are needed to support reliable and effective practice while performing a specific task in the development process. Equally important is the demand for a conceptually integrated suite of methods that can interlock like pieces of a puzzle in supporting the entire development process. Consistently good

performance of a specific task (e.g., information requirements definition) has been the primary role of previous individual methods applied in a stand-alone mode. However, the full extent of a method's potential can only be realized when applied with other methods, each highlighting different perspectives of the problem. In this way, one goal of concurrent engineering — considering more life-cycle factors in the initial design — is achieved while generating new levels of enterprise integration, flexibility, and responsiveness. With this goal in mind, the IICE program has given high priority to developing a conceptually integrated suite of methods.

Two main tasks, established as part of the Methods Engineering Thrust plan, outline the primary focus of these efforts. The first task sought to develop the foundations for an engineering discipline guiding the appropriate selection, use, extension, creation, and integration of methods. That is, the goal was to develop methods for "engineering" methods with predictable effectiveness, and to develop techniques and metrics for analyzing methods. The second task sought to identify and fill method voids. Guiding the scope of new methods development was a focus on the factors needed to develop and evolve an information-integrated concurrent engineering enterprise successfully. Together, these two primary tasks not only satisfied short-term needs for methods, but also provided the foundations for future reliable method development activity.

Among the key activities undertaken through the Methods Engineering Thrust are the following:

1. Develop foundations for a methods engineering discipline.

2. Develop new IDEF methods to fill method voids.

3. Develop, apply, and test candidate method formalization approaches as a foundation for future standards development.

4. Explore method integration and model translation strategies.

5. Explore method classification schemes to aid the analysis, comparison, selection, and extension of methods.

Significant Accomplishments

The accomplishments of the Methods Engineering Thrust can be described in terms of the different types of users. The most obvious beneficiaries of Methods Engineering Thrust activities are the users of the individual IDEF methods. For this class of user, the IICE effort has produced five methods that fill previously identified voids. These methods are the Process Description Capture method (IDEF3), the Object-Oriented Design method (IDEF4), a specialization of IDEF4 helping users specifically target their designs for implementation using the C++ object-oriented language (IDEF4/C++), the Ontology Description Capture method (IDEF5),and a prototype Business Constraint Discovery method (IDEF9). Direct users of these methods include corporate decision makers, systems analysts, consultants, knowledge engineers, systems designers and developers, and project managers. Technology developers are leveraging Methods Engineering Thrust results to enable and/or accelerate the development of

different products. For example, the method formalization standard developed for use in engineering the IICE IDEF methods has been adopted by both the Institute of Electrical and Electronics Engineers (IEEE) and Federal Information Processing Standards (FIPS) committees responsible for developing standards for the IDEFØ Function Modeling and IDEF1X Semantic Data Modeling methods. Direct technology suppliers now offer products that are direct implementations of Methods Engineering Thrust products. For example, two software vendors have developed automated tools supporting application of the IDEF3. The key accomplishments realized through the Methods Engineering Thrust are summarized below.

1. Completed a study of historical patterns associated with the emergence of new methods and formulated a road map for evolving current methods engineering practice toward its maturation and acceptance as an established engineering discipline (Knowledge Based Systems, Inc. [KBSI], 1991).

2. Defined a general approach to method engineering that has been applied and refined through its application to the development of IICE IDEF methods.[1]

3. Developed a unique approach for balancing the need for expressive power in a method language and the need to focus attention on the key types of information that have direct relevance to the task served by the method. The approach is based on the concept of a *family of methods and supporting method languages* integrated at a logical rather than physical level.

4. Explored method needs and requirements to support evolution toward an information-integrated concurrent engineering (CE) environment. From these findings, a basic set of methods was proposed to satisfy those needs.

5. Produced four mature additions to the IDEF family of methods (IDEF3, IDEF4, IDEF4/C++, IDEF5,) and completed initial developments on those listed below.

    a. Design Rationale Capture method (IDEF6).

    b. Human-System Interaction Design method (IDEF8).

    c. Business Constraint Discovery method (IDEF9).

    d. Network Design method (IDEF14).

_____

[1] An overview of this approach is provided later in this chapter under the heading, "The Method Engineering Process."

6. Conducted successful technology transfer and technology transition activities involving the emerging IICE methods. Some of the most significant accomplishments in this category are listed below.

   a. Published Armstrong Laboratory technical reports describing IDEF3, IDEF4, IDEF4/C++, and IDEF5.

   b. Generated a large core of trained IDEF3 users at Oklahoma City Air Logistics Center (OC-ALC), resulting in IDEF3's use on nearly 70 process mapping and improvement projects.

   c. Presented eight conference papers describing the emerging IICE methods and their potential to audiences at the IDEF Users Group, Continuous Acquisition and Life-Cycle Support (CALS-CE), and Factory Automation (AUTOFACT) conferences.

   d. Received and fulfilled invitations to conduct the Research Forum at two conferences of the IDEF Users Group.

   e. Successfully published chapters describing the products and lessons learned through Methods Engineering Thrust activities in *Knowledge Base Systems in Design and Manufacturing* (Kuziak & Dagli, 1994) and the *CIM Implementation Guide* (Bertain, 1991).

7. Developed a standard approach for method formalization that is being used internally on the IICE project and externally toward the development of FIPS and IEEE standards for IDEFØ and IDEF1X.[2]

8. Developed a method meta-language consistent with the Neutral Information Representation Scheme (NIRS) concept, laying the foundations for automated method language translation.

9. Leveraged the IICE program's method and model integration research to assist in the development of the Semantic Unification Meta-Model[3] (SUMM) standard, with

---

[2] For more detailed information on the results of items seven through ten, refer to the Integrated Systems Theory Thrust section of this report.

particular contributions to the SUMM's emerging dynamic information component. The SUMM standard is sponsored by the international Product Data Exchange using Step (PDES) initiative. Model representation and integration needs identified through the Methods Engineering Thrust will establish the foundations for a standard, uniform, theoretical framework to rigorously define modeling languages and the rules for integrating models based on those languages.

10.   Characterized three levels of model translation (transliteration, production rule translation, and ontology-driven translation) that can be applied to the challenge of translation among varying forms of representation languages, including the IDEF method language. Completed the mathematical foundations for implementing ontology-driven translation, and successfully demonstrated the feasibility and limitations of both the transliteration and production rule translation strategies in and across different method types (e.g., Data Flow Diagrams to IDEFØ diagrams and IDEFØ diagrams to IDEF1X diagrams) (Mayer & Wells, 1991).

Background on Methods

Nature and Importance of Methods

The importance of methods has long been recognized in the manufacturing industry. Methods are prominent in the "5 Ms" of manufacturing: manpower, methods, materials, machines, and money. During World War II, KRUP Industries provided much of the machinery of the German war effort. When it became obvious that Germany would fail, KRUP Industries demonstrated powerful foresight by moving its critical people and records to safe places in Switzerland and Austria. By 1952, KRUP was once again a thriving industry. KRUP understood that materials, machines, and money can be replaced, but manpower and methods are the vital organs of the industry.

The history of a method development, if it could be captured, would clearly demonstrate that the method was not developed arbitrarily; yet methods tend to be developed piecemeal by discovering what works well in a domain. The underlying scientific reasons for why a method works may not be understood by the discoverer. Rather a hunch, intuition, or accidental circumstance may lead to such discoveries.

---

[3] The PDES/SUMM standard is being developed as the international standard for model language specification, model data interchange, and model data interpretation. This standard is commonly referred to as the Semantic Unification Meta-Model.

Because the scientific basis for a method may be unknown, methods tend to be difficult to enforce. That is, it is difficult to convince someone to use a method for which no logical reasons can be given, as is demonstrated in the following anecdote from a railroad construction yard.

A new welder was assigned the job of forming and welding stays for the large wooden barrels used as containers on the railroad cars. Proud of his welding prowess, the newcomer chafed at the foreman's insistence in giving instructions, at great length, of exactly how to perform the job — instructions which the young welder considered outdated. He argued with the foreman that he could do the job much faster his own way. The foreman, for his part, insisted adamantly that the job must be done just this certain way or it would not be done right.

Unconvinced, the welder decided to show the foreman. In two days he completed a week's worth of work, which he triumphantly showed the foreman. Wordlessly, the foreman took one of the stays in hand, climbed to the top of the water tower, and threw the stay to the ground. The welder watched as the stay bounced twice and shattered at the seam. Climbing down from the water tower, the foreman quietly handed the welder a stay built the foreman's way and pointed to the tower. When the welder repeated the experiment with the foreman's stay, the stay hit the ground, bounced repeatedly, and held firm.

The young welder may never understand why the foreman's method works, but he will undoubtedly use it in the future. the foreman's method is typical of methods, in general, because it represents "best practice" in the world of welding stays — a best practice learned through experience over time.

Methods encapsulate best practice to accomplish a given task together with a specialized language of expression. For traditional manufacturing methods, the language of expression often takes the form of highly specialized terminology. For example, welding methods require a highly specialized language to describe heat, bead, depth of penetration, and so forth. In this example, the language of expression assumes a far less visible role in the method. Among methods used for traditional architecture and information systems development, however, the language of expression generally assumes an added dimension — one of graphical depiction. Since graphical language extensions supporting the method become its most visible component, graphical languages supporting methods are often mistaken for the methods themselves.

Methods provide the rules for success and the structure defining the boundaries of reliable application. Graphical languages, when used, not only focus the practitioner on a limited portion of the overall problem domain but also highlight key information so it can be extracted easily. In this way, methods attempt to carve the world into discrete, manageable chunks while sensitizing the practitioner to important uniformities that would normally require years of experience to recognize.

Methods are designed to filter out irrelevant or unneeded information for the task. This feature makes it difficult to express things outside the presumed domain of application. The language facilitates a method that tends to obscure those things that may be of interest in a different context while simultaneously highlighting what is directly relevant.

How Methods Have Evolved

Most successful methods evolve over the course of years, from informal practice to more well-defined, formal practice. The beginnings of independent methods generally can be traced to a recurring need or problem. Individuals' intuition drives the development of practices aimed at solving such problems. This process often results in several approaches. From this group of options, a dominant practice eventually emerges. Those organizations that survive can often attribute their success to individuals who were willing to champion the new methods, often at the risk of their reputations and/or jobs. During this phase, the method practice begins to solidify. Successful individual practice begins to be recognized for providing the business with a competitive advantage. This motivates the enterprise to formalize the method as a standard business practice. With continued application, refinement, and proliferation throughout industry, the method eventually evolves into a standard — the industry-wide best practice. As more widespread attention focuses on the method, universities recognize it as a valuable tool and include it in their curricula.

Interviews with method developers added additional insight to how methods are developed. From their perspective, three distinct phases of development occur. During the first phase of method development, the Foundation Building Phase, the motivations behind the method are documented, and the needs, potential benefits, and potential cost are defined. Then, the experience base is studied and verified, people in the domain who exhibit the best practice (as judged by their peers) are interviewed and studied, and the documentation developed by these experts is studied. In the second phase, the Focus Phase, existing methods that provide some, or possibly most, of the desired support are used and the set of best practice concepts are identified. The adequacy and consistency of the basic concepts are set by formalizing and testing them. The third phase is the Physical Design Phase. Expert method developers found it difficult to explain the procedure they follow to produce a physical design for the method.

As with any design activity, the highly iterative nature of method synthesis, innovation, testing, and modification cannot be reduced to a simple process. However, it is possible to characterize the key thought processes, most significant design considerations, and their interrelationships. Formulating such a characterization was one of the tasks undertaken by this thrust

Components of a Method

A method may be thought of informally as a procedure for doing something that may or may not be accompanied by a representational notation. Methods may be more formally described as consisting of three components (Figure 1): a definition, a discipline, and a use. The method definition is established by characterizing the method's basic motivations, concepts, and theoretical foundations. The definition component is developed by method developers familiar with methods engineering principles (e.g., formal language design, method ontology definition). The discipline component includes the syntax of the method and the procedure by which the method is applied. Many methods have multiple syntaxes which have evolved over time or which emphasize different concepts within the scope of the method. From a method engineer's standpoint, the discipline component is the user interface for the method. This component is often the only one presented to typical users. The use component characterizes how to apply the method in different situations, such as when the method is applied together with other methods versus in a stand-alone fashion.
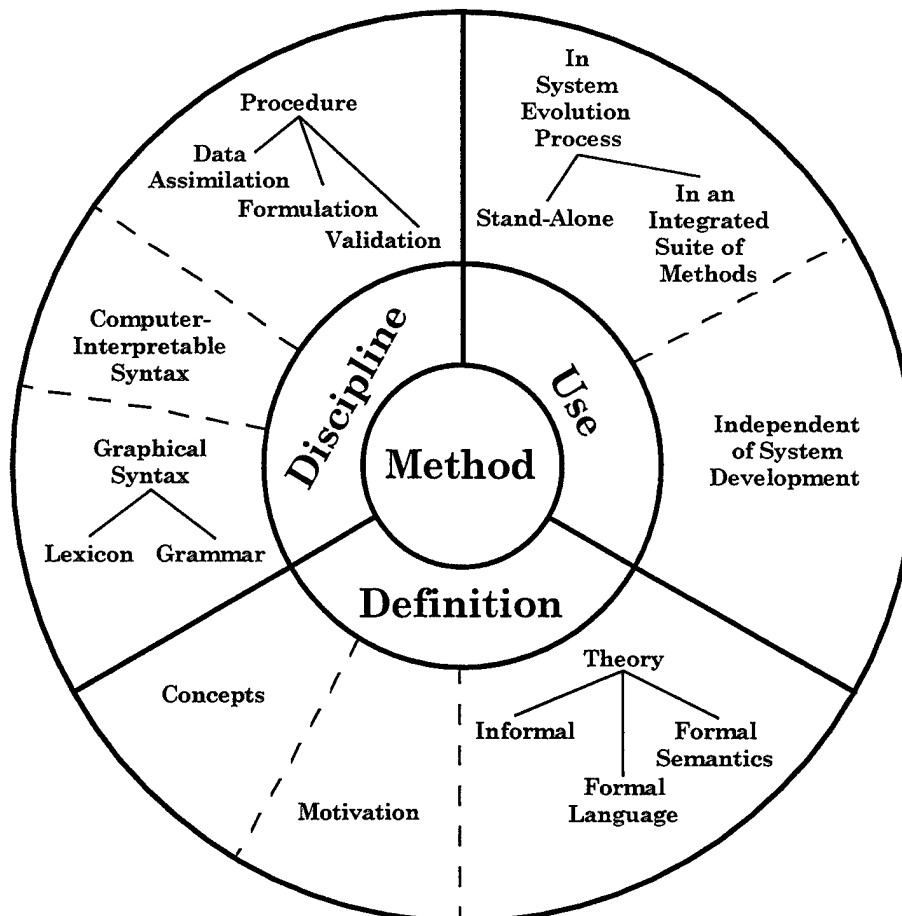
Figure 1

Anatomy of a Method

Obstacles to the Advancement of Methods

An IICE survey conducted among method developers indicates that a method may take as long as twenty years to gain widespread acceptance and use. Some of the most significant obstacles to method acceptance include a lack of engineering discipline in developing methods, lack of familiarity with the nature and potential of methods, and the lack of automation support for methods.

Lack of Engineering Discipline in Developing Methods

A general lack of formal training among method users and developers generates confusion over how methods should evolve. This confusion can be attributed partly to lack of familiarity with formal languages, finite state machines, semantic models, and other relevant theoretical foundations. For example, during early efforts to stabilize the IDEF methods, many IDEF users believed an IDEFØ meta-model[4] of the IDEFØ method could serve as its formalization. Few could see that this proposed exercise would be like defining an unknown word with itself. A similar proposal, using IDEFØ to formalize IDEF1X, could be compared to defining one unknown word with another unknown word. Without the benefit of formal training, individuals found these proposals intuitively appealing and logical. Unfortunately, other ill-conceived practices to change existing methods or to develop new ones are not always easily combated. For this reason, the advancement of a methods engineering discipline, and its recognition as a legitimate field of study, continues to be an uphill battle.

The method development process must be systematized so that it can become widely accepted and practiced. The contributions of the IICE program in this area could lead to improvements in the quality of emerging methods and to a growing acceptance of methods by the academic community as an area of study, eventually giving rise to the discipline of methods engineering as a new branch of systems engineering.

Lack of Familiarity with the Nature and Potential of Methods

Lack of familiarity with the *nature* and *potential* of methods creates more obstacles to their widespread acceptance and use. By their nature, methods provide leverage for a restricted class of problems. For all other problems, they may simply get in the way. Methods are designed to work well when applied correctly. If applied incorrectly, however, the results of method application are left largely to chance. Those unfamiliar with the nature of methods may select an inappropriate method for their needs. Or they may inappropriately tailor or misapply the method while trying to make it fit their needs. Others may unrealistically expect methods to eliminate, rather than facilitate, steps that must be taken in systems development. Lack of awareness about the potential of methods also creates obstacles to widespread acceptance and use. These obstacles do not arise from poor results, but from

---

[4] A meta-model is a model of the semantics or meaning of a given method's modeling language.

13

achieving only limited benefit from method application. For example, most users of the IDEF methods find value in using them to understand the domain of interest. When a method is used exclusively for this purpose, that is likely the only benefit that will be recognized. This should not, however, lead one to assume that furthering one's understanding of the domain is the only benefit of method application.

The following observations may help users understand the nature and potential of methods, and thereby help users derive more benefit from them.

1. **Methods target specific problem domains.** Each IDEF method targets different problem domains and supports correspondingly different groups of users. Although a particular group of users may be interested in becoming proficient with methods that support other specialties, there is no inherent requirement to do so.

2. **Methods reduce risk — at a cost.** There are inherent risks assumed by choosing not to apply a certain class of methods (e.g., information requirements definition methods, process knowledge capture methods, technology-specific design methods) just as there are costs (e.g., labor, training, and tool costs) assumed by choosing to use one. Decision-makers must strike a balance between affordable cost and acceptable risk when choosing which methods to use.

3. **Methods help reduce life-cycle development costs.** Applying IDEF methods may involve higher up-front investment but generally will produce higher quality results and an overall savings in time and money in the course of the project life-cycle.

4. **Following procedural guidelines is important.** A principal cause of poor results is a failure to follow the procedural guidelines of the method. Guidelines are often disregarded because they are mistakenly believed to be either arbitrary or rigid. To the contrary, these guidelines are not arbitrary. They emerge as the most successful practices used by experts are observed and recorded. Using these procedures can have a positive impact on the quality of results. Conversely, these guidelines are not rigid. Procedural guidelines may be adapted to accommodate variances in project size and team composition. However, care should be taken to ensure the essential elements of the method procedure are recognized and preserved.

5. **The IDEF methods are scalable.** IDEF methods can be adapted for application on both large- and small-scale development efforts. Although the procedural guidelines for the IDEF methods generally assume a large-scale effort, users can make appropriate adaptations for smaller efforts.

6.   **Methods help organize large volumes of information.** IDEF methods provide efficient mechanisms to help manage the large volumes of data generated by analysis and design activity. This is particularly valuable for projects involving large teams needing to share information.

7.   **Methods provide benefits to users and customers.** Users who directly apply the IDEF methods may derive the most obvious benefit from them, helping them understand the environment and complete assigned tasks with consistent results. Customers, who sponsor the use of methods, may realize additional benefits — if the methods are applied correctly and maintained in an easily reusable form. For example, intermediate products of method application may help customers rapidly determine the state of a design or review the collective knowledge of those interviewed on a particular subject.

8.   **Methods provide a common language of communication.** The IDEF methods are accompanied by special-purpose languages with clear syntax and semantics. This feature of methods is valuable in establishing a common basis for communication about relevant aspects of the project.

9.   **Different methods can "talk" to each other.** While methods can be used effectively to promote integration through information sharing and reuse in a domain, they do not ensure information sharing and reuse across domains. Users often express the need for information sharing across domains by saying that methods should be able to "talk" to one another other. That is, users would like to reformat easily reusable information captured through the application of one method to accelerate progress in completing the work supported by another method. When methods are applied independently, without first deciding how the information should be shared, only limited integration will be achieved. Integration of effort can, however, be achieved by determining early what information will be collected and what needs to be shared. This information can then be used to establish the modeling conventions to maximize information sharing and reuse among project participants using different methods.

10. **Complementary methods provide added leverage.** A machine shop owner would not be expected to dismiss lathes and their corresponding machining methods simply because he already has, and is familiar with using, a milling machine. Each machine is suited for different needs. For some jobs, only one machine is needed. For others, both are needed. Methods are no different. Decision-makers may limit the potential leverage that can be obtained from methods by restricting the range of methods that can be applied on a project. Policies requiring the use of a restricted set of methods may make it difficult to address unique aspects of the project efficiently. They may also create problems by

causing an inadvertent loss of information. One way information can be lost is if the chosen method is not equipped with language facilities to capture and explicitly represent information that may be of importance. Information can also be lost by using non-standard conventions created to capture information that cannot otherwise be captured explicitly. These conventions, if not standardized, soon make it unclear how to interpret the information represented in the chosen method language correctly.

Lack of Automation Support

Lack of needed automation support for methods also creates obstacles to their advancement. Software tools that support individual methods help increase efficiency and encourage uniform adherence to the supported method. More recently, methods automation research has explored the development of modeling environments that support the *integrated* application of multiple methods. This capability is critical in overcoming problems caused by stand-alone tools.

Some problems that can be overcome by more sophisticated method support tools and environments are listed below.

1. Inaccessible information captured in differing method languages and tool formats.

2. Lost or uninterpretable information caused by method misapplication.

3. Limited ability to reuse models.

4. Unnecessary duplication of effort.

Many of these difficulties could be solved with an environment in which information created in one method can be used successfully to assist in the creation of other models. Environments supporting *automated model translation* would establish foundations for casting existing data into new contexts, thereby yielding new information.

Though conceptually simple, the envisioned translation environment is not easy to implement. Some issues and questions that emerged while investigating the development of such an environment are as follow.

1. How fully can the translation between models be automated?

2. Are the resulting models useful when populated from other methods?

3. What can be done to handle methods that contain different kinds or amounts of information?

4. Is the intent of the method compromised when it is translated from another model?

16

5.   How can the configuration of dependency links (particularly accountability and design rationale) across model types be managed efficiently?

Despite the difficulties, the development of automated model translation environments has greatly enhanced efforts to realize an Evolving Information-Integrated System (EIIS) successfully. This strategy has been investigated as a coordinated effort between the Methods Engineering, Integrated Systems Theory, Ontology, and Experimental Tools Thrusts.

Evolving Toward a Methods Engineering Discipline

Methods are themselves systems that exhibit qualities analogous to any system that is engineered. Methods take inputs and convert them into products. They must be "installed," operated, and maintained. Their application, or "operation," comes at a cost in time and money. They exhibit varying levels of reliability in the way the support user performance. Different methods also exhibit varying levels of quality in the products that result through their application.

As systems, methods can be designed to exhibit desirable characteristics. Some of the desired characteristics might include ease of use for the class of individuals supported, low application cost, low variability in the quality of product produced through method application, ease of inter- and intra-method data integration, and so forth. These characteristics, however, are consequences of good method design. They do not simply happen by chance. Today, no recognized engineering discipline exists to guide method developers through a reliable, scientific approach to method design. The mathematical foundations, engineering tools, and design techniques that could be used to evolve such a discipline was a subject of investigation in this project.

Investigation into a methods engineering discipline began with an exploration of existing methods engineering practices. The Methods Engineering team then formed a strategy to bring engineering discipline to methods development. The following information briefly describes the progress made toward that goal.

What is an Engineering Discipline?

An engineering discipline may be defined broadly as a set of rules, methods, and practices used to solve a certain class of problems.

To fully understanding how engineering disciplines evolve, the Methods Engineering team surveyed a number of established engineering disciplines (e.g., Operations Research, Civil Engineering, Industrial Engineering, Mechanical Engineering). While doing so, some common elements surfaced. Among the characteristic elements possessed by established engineering disciplines are the following:

17

1.   A well-developed nomenclature.

2.   A well-developed set of theories.

3.   Characterization of "best practice."

4.   A reasonably well-defined application domain.

5.   A set of general design principles.

6.   The intelligent use of understood idealization techniques, such as mathematics, statistics, philosophy, human factors, and so forth, to perform modeling activities.

Engineering disciplines draw on these elements, together with a knowledge of the natural sciences, to conduct engineering research — research involving experimental activities as well as modeling activities. From such efforts, predictive models are created, new phenomena are discovered and explained, and process technology and new materials are developed.

Given this characterization of engineering disciplines, it was concluded that these same elements must be present for methods development practice to be elevated to the status of an engineering discipline. Using the six characteristics of recognized engineering disciplines, we determined that the current methods development practice is at a relatively immature state. For example, established engineering disciplines have not generally been the source of methods like the IDEFs. A number of people from both industry and academia who have tracked or been involved with methods during much of their careers were interviewed to obtain their assessments of the current state of methods development practice. The majority of those interviewed believe there is a prevailing lack of engineering discipline and a need to develop that discipline.

Method Engineering Process

Interviews with method developers, coupled with the experience gained through IICE IDEF method development, enabled more refined descriptions of the method engineering process. Figure 2 illustrates a high-level characterization of the process for engineering a method. Knowledge engineering activities assume a central role in this process because methods emerge largely through efforts to embody expert sensitivities, rules of thumb, strategies, and experience in a form that can be applied successfully by less-experienced individuals. There is also a great amount of interaction among the processes involved in engineering new methods. For example, the discovery of a new procedure, strategy, or heuristic can instigate radical changes to the graphical syntax of a method. Likewise, an innovative change to graphical syntax may drive changes to the method procedure. Additionally, a number of changes can come from the practical application of the method. Changes that simplify its use or alter its scope of application may be made.

18
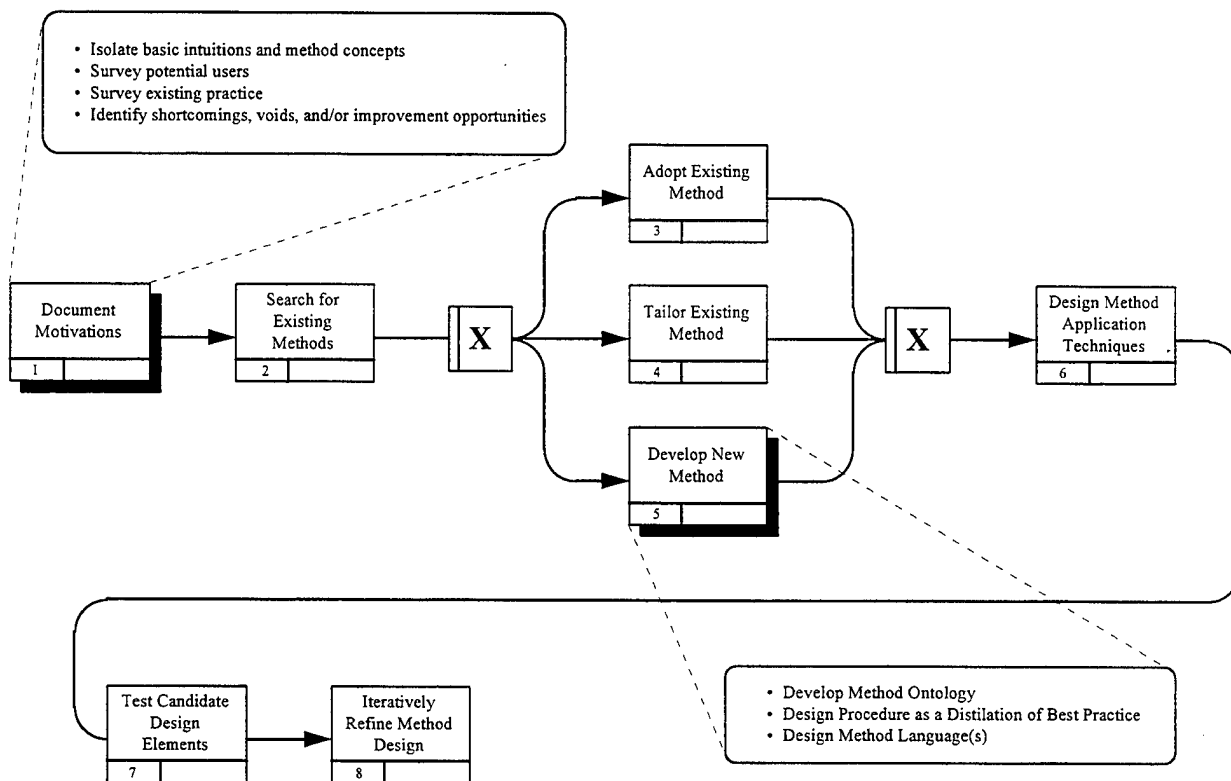
Figure 2

Description of the Method Engineering Process

Method Formalization

A complete and rigorous formalization is pivotal to the successful development, comparison, selection, and integrated application of methods. Typical method users generally will not perceive the need for a formalization until they have struggled to understand or provide automated support for a method, or to integrate a team using multiple methods. Interestingly, when the need is recognized, many people in industry envision an adequate formalization as a graphical syntax tied to an imprecise conceptual view of the method. Despite a growing awareness of the need for formalization, little consistent guidance has been available to those needing to accomplish the task.

Formalization begins where many attempts stop — with a concise, common-use characterization of the method. The "common-use" characterization includes a clear statement of the following elements.

1. *Motivations* behind the method.

2. *Historical experience* on which the method is founded.

3. *Components* of the method including:

19

a. Basic concepts.

b. User language syntax (graphical or textual) and informal semantics.

c. Kind of information the method is intended to discover or manage.

d. Techniques for applying the method.

e. Use procedures of the method (e.g., when to use it and what to do with the results).

Although the common-use characterization lacks the mathematical and logical rigor of a formalization, its development assists downstream formalization. Common-use characterization efforts help develop an informal description of the user language syntax and intended semantics. Additionally, as formalization efforts progress, the common-use characterization helps answer questions through insights and discoveries made from observing the method in practice.

Having completed the common-use characterization, one can develop a "mathematical" formalization of the method language. In developing this formalization, a first-order language is designed, based on the common-use language of the method. The first-order language provides a convenient mechanism for designing a mathematically sound, model-theoretic semantics for the method. Constructing a first-order language equivalent to the common-use language helps detect ambiguity and incompleteness in the common-use language, and may be used to assist developers in method support tool requirements development and design. Following language design, a mathematically sound model semantics, generally based on set theory, is designed for the formal language. After the model semantics has been validated against the method language, it can be used to determine characteristics and properties of the original method using proof-theoretic techniques. Such models have been successfully used to predict "meaningless" statements in the original method language that may otherwise escape detection by method developers and practitioners.

An extension to the mathematical formalization that supports automated model interpretation and model data integration is the "ontological" formalization. Its focus is on the intentional uses of a method. This type of formalization characterizes the connections between the real-world and statements about the real-world in the language of the method. This differs from the second level in a subtle but important way that is best explained by example.

Mathematical formalizations for the IDEF1 and IDEF1X languages provide a rigorous foundation for testing the grammatical correctness of statements made in either language. As such, they also provide a clear specification for tool developers attempting to provide automated support for stand-alone application of the methods. However, having the mathematical formalizations does not enable one to determine whether an IDEF1X entity is identical to an IDEF1 entity class. Mathematical formalizations generally are not able to state that IDEF1

20

entities represent the information kept about real-world objects, while IDEF1X entities are generally used to represent the real-world objects themselves. Thus, an IDEF1 entity labeled "F-16" stands for the information that is managed about F-16s, whereas an IDEF1X entity labeled "F-16" stands for the set of F-16 aircraft. These are very different notions. The ontological formalization is an attempt to characterize these intensions and use connections associated with a method. whether an IDEF1X entity is identical to an IDEF1 entity class. Such a formalization appears necessary to make significant progress in automated model translation. As illustrated by this example, the semantics, or meaning, associated with entities becomes vitally important when concerned with model interpretation.

When formalizing a method, the general approach is as follows.

1.   Analyze model examples of the method.

2.   Develop a syntax for representing information in the examples.

3.   Develop appropriate semantics for the syntax.

In reality, this sequence is applied iteratively since there is some amount of recursive testing of the syntax and semantics. The formalization process often uncovers a number of inconsistencies in the syntax and semantics. These inconsistencies often occur when language development is undertaken without a well-stated and well-understood theoretical foundation upon which to build the syntax and semantics. The NIRS provides the needed foundation for defining the formal syntax and semantics of a method language.

Over the course of its evolution process, a method may undergo one or more of the formalization steps described above. Investments in formalization activity may be directed largely toward defining requirements for automated method support tools. Equally important, method formalizations aid in transitioning successful methods to new generations of practitioners while encouraging correct use and enabling effective model data reuse and sharing.

However, neither software support nor formalized representations constitute a method by themselves, nor do they carry out the intended purpose of a method. Just as shovels do not dig holes but provide leverage to the person digging, methods are designed to leverage the human mind. While a method is designed to assist and motivate human intellectual activities, it will not make decisions, create insights, or discover problems. Methods must be recognized as a means to gain leverage rather than as a mechanism that eliminates the need to apply effort. Yet, without formalization activity the leverage that can be achieved through method application will be limited. Formalization increases the potential benefits of methods by enabling reliable and consistent application, by accelerating the speed and quality of automated tool development activity, by facilitating information integration, and by promoting reuse of method application results to support other system analysis and design activities.

Foundations for Information Exchange Between Methods

Method integration efforts have long recognized the need for a common language that enables information exchange across models of the same type as well as among those of different types. Traditional attempts often begin by establishing one method language as the meta-language for modeling each method to be integrated. Once the meta-modeling language has been selected, meta-models of each method are constructed in the standard meta-language. Finally, attempts are made to use the resulting meta-models as the intermediate step to inter- and intra-model data exchange.

This process, although intuitively appealing, has a number of problems. For example, it is difficult to compare the relative worth of one method language to another. Expressive power, a commonly adopted metric for selection, is not necessarily a good measure of merit because method languages are designed to overlook irrelevant information. Such a situation is similar to comparing the relative merits of an information collector tuned to collect visual information (an eye) and an information collector tuned to collect auditory information (an ear). One cannot assume that because an ear cannot "see" it is less valuable than an eye.

One could, for example, choose the representational apparatus of sight as the standard meta-modeling language. When using this representational medium to capture information collected by a method attuned to auditory information, the resulting meta-models perform exactly as expected; they capture a highly specialized slice of information about the method being modeled and fail to capture what is not within the "visual range" of the standard language. Not surprisingly, the third step — using meta-models of each method in the standard meta-language to exchange information — results in only limited success.

The neutral representation must capture many different forms of information collected by multiple, highly specialized information receptors — that is, methods. Furthermore, the neutral representation language will not be a method language.

For example, assume a company uses only one method and its only interest is in enabling data exchange between models constructed from the standard method. Can the method language itself be used as the neutral representation language? Will this action, by definition, enable model integration across the enterprise? Surprisingly, the answer is "No" because models actually carry with them more information than is explicitly stated in them. Although somewhat counter-intuitive, this phenomenon has frustrated method integration attempts through standardization for many years.

For instance, imagine a method language that can state, "John is a Democrat and John is not a Republican."[5] This statement takes the logical form "$Pa \& \sim Qa$." Further examination reveals that this logical statement carries with it the information that $P \nless Q$ (i.e., that P is not a subtype of Q [since something that is of type P is not of type Q, namely, a]). In other words, it can be inferred that Democrats are not types of Republicans. This example may appear trivial, but it illustrates the presence of a structural constraint that constitutes information conveyed by the model but not directly expressible in the language.

This example shows that the neutral representation language must be capable of capturing both what *is* and what *is not* explicitly expressible in the method's language. Without such capability, even integration between models developed using the same method is severely restricted.

NIRS can serve as the formal foundation for inter-model data transfer and translation. It is a "lowest common denominator" representation of information collected by the IDEF methods and is based on a powerful theoretical foundation known as *situation theory*. By using this foundation, NIRS can represent precisely the information types of a method. Furthermore, since NIRS provides a common ground for all IDEF methods, it is ideal for formally integrating the IDEF methods.

NIRS includes the following key characteristics:

1. NIRS possesses a language and semantics, yet it is neutral (i.e., not biased toward any one method). It has not been tailored to a specific application, context, or job, nor is it confined to one type of situation.

2. NIRS supports integrated application of a suite of methods and method-specific modeling languages for the large number of differing tasks and user/developer roles associated with EIIS development.

3. NIRS has the capacity to express temporal information.

4. The formal NIRS ontology includes the basic elements of situation theory.

5. NIRS is able to represent all IDEF methods. Models produced in any IDEF method can be expressed in NIRS.

6. NIRS is capable of representing information that is implicit in the method languages.

---

[5] In this example, we describe a method whose language has the expressive capacity of a monadic, first-order language without quantifiers or variables (i.e., essentially, a language which only captures names of objects, simple predicates, and Boolean operators).

Meta-Language Developments

In addition to developing the formal foundations for intra- and inter-method information exchange (which was accomplished through definition of NIRS), an implementation of the NIRS meta-language concept has been developed. This implementation has been named the Information System Constraint Language (ISyCL).

A meta-language is needed to state the design of a method language. This meta-language uses situation theory as a framework for providing a rigorous syntax and semantics. This framework provides language designers with the guidance necessary to allow for automated inter-model data transfer and translation. Thus, this theory should be viewed as the structure for an information model data exchange specification. This theory also serves as the first step toward achieving an integrated environment for evolving an EIIS.

The notion of a general purpose constraint language evolved from previous Air Force work that began as an effort to define a constraint specification language for the IDEF1 modeling method (Decker and Mayer, 1992). The emerging language structures were similar to those investigated for the IDSE conceptual schema representation language and for NIRS's potential use as an evolving system description capable of supporting automated knowledge-based model translation. The theory developed from this work serves as the formal foundation for a family of languages supporting automated knowledge-based model translation. Further details about the NIRS can be found in the appendices.

IDEF Method Developments

In addition to providing guidelines for selecting, extending, and creating methods, the IICE Methods Engineering Thrust set out to identify and fill method voids. Methods for developing the information integration mechanisms required to support CE activities were of particular interest. New methods development was confined to those previously unavailable. IDEF3, IDEF4, and IDEF5 were developed in direct response to the needs identified for an information-integrated CE environment.

Additionally, these investigations demonstrated the need for a family of methods, with each devoted to unique cognitive tasks. The IICE methods engineering team realized early there was a need to distinguish between analysis and design activities. Each involves entirely different cognitive processes supported by correspondingly different method design strategies. In a similar fashion, exploration into new method needs uncovered the necessity of distinguishing modeling and description capture; the differing focus of these activities required different method design strategies.

Models Versus Descriptions: One contribution made by the IICE program to the advancement of methods engineering was the establishment of clear design goals and distinctions between methods aimed at developing

models and methods aimed at collecting descriptions. Understanding the importance of this contribution requires discussion of the distinction between models and descriptions.

People tend to use the terms *description* and *model* loosely in ordinary language. When these terms are used in the context of IICE project findings, however, they have a precise technical meaning. The term description is used as a reserved technical term to mean records of empirical observations; that is, descriptions record knowledge that originates in or is based solely on observations or experience. The term model is used to mean an idealization of an entity or state of affairs. That is, a model constitutes an idealized system of objects, properties, and relations that is designed to imitate, in certain relevant respects, the character of a given real-world system. Frictionless planes, perfectly rigid bodies, the assumption of point mass, and so forth are representative examples of models. The power of a model comes from its ability to simplify the real-world system it represents and to predict certain facts about that system by virtue of corresponding facts within the model. Thus, a model is a designed system in its own right. Models are idealized systems known to be incorrect but assumed to be "close enough" to provide reliable predictors for the predefined areas of interest within a domain. Despite modeling accuracy, the true benefit of models stems from the speed and low cost with which relevant aspects of a real or proposed system can be evaluated. However, the usefulness of a model is limited to the range of questions addressed by its design and the reliability of its approximations in differing contexts.

A description, on the other hand, is a recording of facts or beliefs about something within the realm of a domain expert's knowledge or experience. Such descriptions are generally incomplete; that is, the person giving a description may omit facts that he or she believes are irrelevant, or which he or she has forgotten in the course of describing the system. Consequently, descriptions are not constrained by idealized, testable conditions that must be satisfied, short of simple accuracy.

The purpose of description capture may be simply to record and communicate process knowledge or to identify inconsistencies in the way people understand how key business processes actually operate. By using a description capture method users need not learn and apply conventions forcing them to produce executable models (e.g., conventions ensuring accuracy, internal consistency, logical coherence, non-redundancy, completeness). Forcing users to model requires them to adopt a model design perspective and risk producing models that do not accurately capture their empirical knowledge of the domain.

Description capture may also be undertaken to produce models. Whether accomplished implicitly or explicitly, descriptions are the raw material from which models are made. Thus, the utility of descriptions may also be realized through their reuse in constructing multiple idealizations or models.

Interestingly, models are a form of description. The reverse, however, is not true. A description is not a model. Models are exercised to create analysis data that is not available in descriptions. Unlike models, descriptions do not create analysis data; they may, however, serve as one form of analysis data. For example,

descriptions of bus routes and arrival times may be useful forms of data for developing a model of the public transportation system but they do not themselves constitute that model. Similarly, descriptions of an automobile, while potentially useful for other purposes, cannot be used to generate finite element analysis data.

Early IDEF methods focused primarily on modeling, whereas the IDEF methods emerging from the IICE program address both modeling and description capture needs. IICE methods such as the IDEF3 Process Description Capture and IDEF5 Ontology Description Capture methods attempt to facilitate description capture and organization. Others, such as the IDEF4 Object-Oriented Design method, target needs requiring effective modeling mechanisms. Interested readers are invited to see Appendix A for high-level overviews of the most mature IDEF method developments accomplished under the IICE program. Five methods are presented: the IDEF3 Process Description Capture Method (Mayer et al., 1992a and 1995d), the IDEF4 Object-Oriented Design Method (Mayer et al., 1992b and 1995e pending), IDEF4/C++ Object-Oriented Design Method — a specialization of the IDEF4 method targeting implementation of object-oriented designs for the C++ programming language (Mayer et al., 1995b pending), the IDEF5 Ontology Description Capture Method (Mayer et al., 1994), and the IDEF9 Business Constraint Discovery Method (Mayer, Crump, Fernandez, Keen, & Painter, 1995c).

EXPERIMENTAL TOOLS THRUST

This section begins with a brief introduction to the Experimental Tools Thrust. This introduction is followed by detailed descriptions of the concepts and issues of the Experimental Tools Thrust.

Introduction

The goal of the Experimental Tools Thrust is to produce operational concepts and architectures for integrated system development environments, and to develop and demonstrate prototype systems based on those architectures. An integrated system development environment is a computing environment that supports and promotes the sharing of function, knowledge, and data across heterogeneous platforms and throughout the product development life cycle. This definition demonstrates that integration is more than the sharing of data between applications (i.e., single entry of data). Integration should also address the sharing of knowledge/information. The distinction is that data are often entered in the context of a certain situation or application; however, sharing data without sharing this context prevents information integration from occurring. Instead, the data transferred are often misused or misinterpreted. The issues surrounding integrated development environments that address this capability are numerous and complex. Accordingly, the Experimental Tools Thrust has four essential purposes:

- Identify and define the issues related to system integration.

- Identify the requirements of an integrated environment.

- Suggest integration solution approaches.

- Demonstrate applicable integration technology.

By understanding the issues, requirements, and technology associated with integration, organizations can recognize means of evolving toward an integrated development environment.

The Experimental Tools Thrust focuses mainly on the development of an Integrated Development Support Environment (IDSE). The IDSE is an integrated system development environment intended to provide facilities for life-cycle-artifact management and control, system evolution management and control, development process support, and data and function integration. These demands necessitate and will result in an integrated system development environment that resides in a heterogeneous computing environment; has a complete range of integrated system development tools; and can be used to plan, design, develop, and maintain enterprise integrated information system.

In many respects, the Experimental Tools Thrust and the IDSE represent the demonstration arm of the IICE program. The Experimental Tools Thrust represents an effort to implement and demonstrate concepts developed as

27

part of the other IICE thrusts. The relationship of the Experimental Tools Thrust to other IICE thrusts is summarized as follows.

1. Methods Engineering Thrust: Methods developed under this thrust provide the foundation for IDSE component tools. Additionally, information collected in the context of each method is integrated by the IDSE.

2. Ontology Thrust: Ontologies are the core of the ontology-driven information integration architecture of the IDSE Levels 2 and 3.

3. Integrated Systems Theory (IST) Thrust: The IST Thrust has developed the formalization for information representation and integration at IDSE Levels 2 and 3.

4. Frameworks Thrust: The Frameworks Thrust contributed to the requirements and design of the Process Layer of the Level 1 IDSE.

5. Three-Schema Architecture Thrust: The Three-Schema Architecture is a source for IDSE implementation strategies, particularly for the Level 3 IDSE. By recognizing the need to separate business rules from applications, three-schema results led to the inclusion of a constraint propagation engine in the Evolving System Description component of Level 3. Additionally, the recognition that multiple schemas are required to represent a system development environment fully led to the dynamic schema capabilities of the Evolving System Description.

6. Technology Transfer/Applications: The Experimental Tools Thrust provides technology to the Applications and Technology Thrust for demonstration. This effort has focused primarily on the demonstration and development activities at Tinker Air Force Base (AFB).

Significant Accomplishments

The accomplishments of the Experimental Tools Thrust are summarized below.

1. Designed and implemented the Level Ø IDSE. The Level Ø IDSE represents a stand-alone system supporting data translation and artifact management. This system was demonstrated at Wright-Patterson AFB. Additionally, the system was ported to the Macintosh and demonstrated at Tinker AFB.

2. Developed the Framework Processor (Process Layer) concept to monitor, manage, and control the system development process automatically.

3. Developed the Integration Services concept to enable and support the functional integration of software applications and utilities.

4. Designed and implemented the Level 1 IDSE. The Level 1 IDSE represents a distributed environment supporting data and function integration, artifact management, and workgroup management.

5. Published a paper detailing the Level 1 IDSE architecture in the *Journal of Systems Integration*.

6. Designed and implemented the Level 3 IDSE. The Level 3 IDSE represents a global information repository designed to support data change management/propagation and data interpretation.

Overall Philosophy

CE affects all aspects of a company's operation, from engineering and manufacturing to accounting and marketing. Considering the entire life cycle during product development requires that people with expertise from different areas be involved in the development of the product from its conception. The key is to develop a team of individuals from different areas of the life cycle who will be responsible for the product. In large organizations, forming these teams can be difficult because physical barriers such as location often prevent development team members from working together. Therefore, to maximize the synergistic effect of good team players, an organization must provide an environment that allows a group of geographically separated team members to interact as though they were in the same room. The major focus of the Experimental Tools Thrust, IDSE, is to provide such an environment.

Environment Requirements

The design and development of an integrated information system based on CE principles, and capable of evolving as new and improved technology becomes available, is a highly complex undertaking. Analysis has led to the conclusion that effective automation support for CE requires not only task-specific analysis, design, and knowledge delivery tools but also an information environment that improves integration between development tools, coordinates development activities, and supports the tracking of product evolution. Characterizations of these features are described below.

Support Integration Between Development Tools. An integrated set of system development and management tools must be provided. There are currently many automated tools that address different parts of the system development process and have different data and hardware requirements. The IDSE must provide a way for tools (from different vendors) to operate efficiently and uniformly as though they existed in a homogeneous environment. Additionally, because tools have different data formats and hardware requirements, the IDSE must provide a way of translating and communicating data among different machines and tools. The IDSE must provide translators that are able to convert data produced by one tool into a format that can be used by another tool.

29

Coordinate Development Activities. The IDSE should provide mechanisms to coordinate activities supporting product or system development. The IDSE can achieve this by representing the development process and by providing facilities to monitor and control that process. This process view should be supplemented with integrated distribution and sharing of product information. Since the data will likely be distributed among a number of different hosts, the IDSE must provide uniform and transparent access to data repositories residing on heterogeneous platforms and managed by different software applications.

Support Product Evolution. While transparent integration and distribution of information is important, the IDSE must also ensure the integrity of product data. The IDSE must monitor the information system design process and impose automatic and manual controls on the data artifact access and modification. This access control should be coupled with a means of performing configuration management and version control.

Development Approach

The previous section outlined general requirements that should be supported by the IDSE. However, IDSE development proceeded across incremental layers of integration. Each layer resulted in an operational IDSE, supporting varying degrees of integration and using different methods to provide that integration. The different levels of integration supported by these IDSE layers allow for incremental development of components and functionality required to support the IDSE and for demonstration of integration techniques as they are developed. This approach also encouraged experimentation with different types of integration strategies. This experimentation assisted in the identification of requirements for an integrated environment, and provided insights into areas of integration requiring closer examination. The following subsections provide a brief overview of each IDSE layer explored.

Level Ø Integrated Design Support Environment. The Level Ø IDSE represents an integration environment that takes advantage of existing hardware platforms and software tools. This approach supports the idea that some enterprises may not have the interest or the resources to install a full-scale integration environment (Level 3). For those situations, the Level Ø IDSE provides basic artifact management and data integration while minimizing resource requirements.

In keeping with the requirement for minimal resources, the Level Ø IDSE operates as a stand-alone utility and relies mainly on the manual integration of tools and data. As with any development environment, a Life Cycle Artifact (LCA) Repository and Manager is required for data storage and access, version management, and change control. Additionally, this manager provides the functionality necessary to browse the artifact repository and to check out artifact data.

Before artifact data can be checked out, it must be logged into the artifact repository. At Level Ø, this check-in procedure is a manual process. The LCA Manager defines a series of steps that have to be executed for the

30

data artifacts to be properly logged into the artifact repository. Conversely, the data artifact browsing and check-out procedures are completely automated at Level Ø.

Level 1 Integrated Design Support Environment. The Level 1 IDSE incorporates the functionality of the Level Ø IDSE, but adds the functionality for data and function integration. This integration comes from providing access to product artifacts in a distributed fashion. Additionally, function integration is achieved through the introduction of the Integration Services Manager, a mechanism by which tools and utilities can establish peer-to-peer and client-server relationships.

Establishing these cooperative relationships between tools and utilities enhances an organization's ability to leverage existing resources by allowing different tools to work together. For example, when data from one tool is required by another, the client tool makes an integration service request to the Integrated Services Manager (ISM). If that service is an advertised service of another tool operating in the IDSE, the ISM invokes the client tool to execute the requested service. This protocol allows for the access of data maintained in another tool and the execution of functions provided by another tool.

Both the enhanced artifact management and integration services require sophisticated network communications capabilities. If an electronic artifact physically resides on a machine other than the machine at which the user sits, the IDSE should have the ability to transparently access and transfer that artifact to the local machine. Similarly, if a requested service is provided by a tool running on a different hardware platform, the ISM should have the ability to access the remote machine, invoke the tool providing the service, and execute the requested service. As with the artifact transfer, the execution of an integration service should be transparent to the user. This network capability is achieved with a Network Transaction Manager that establishes the protocols and functionality to allow processes running on different machines to communicate and share information.

A process component, which manages and controls the product development process, is layered on top of these enhanced integration and networking facilities. This process layer takes advantage of the networking capability to monitor the evolution and state of product artifacts. By comparing the state-of-the-product artifacts against a representation of the system development process used to produce those artifacts, it is possible for the system to establish the state-of-the-development process. This state is maintained and displayed for the project team members.

Level 3 Integrated Design Support Environment.[6] As with the transition from Level Ø to Level 1, the Level 3 IDSE maintains the functionality of the previous levels. The additional functionality provided at Level 3 centers on fine-grained information management and advanced reasoning with- and interpretation of- that information. In this context, the IDSE moves from maintaining data and artifacts produced as part of the system development process to maintaining the *information* represented by those artifacts. This information is collectively referred to as the Evolving System Description (ESD).

Maintenance of the ESD is achieved by providing fine-grained artifact management. With Levels Ø and 1, relationships and dependencies could only be established between entire models or documents. For example, the design specification could be shown to depend on the requirements document, but relationships between specific design characteristics and the requirements that promoted that design approach could not be established. With the approach taken by Level 3, relationships between individual requirements and design specifications can be established. This functionality provides for a more complete system description by better supporting the capture of design rationale.

The capacity to capture this description is possible through the addition of an object-based repository and the use of a neutral information description language (ISyCL) for moving information into and out of the repository (Fillion, 1995). Each tool wishing to pass data to and receive information from the repository requires an ISyCL Interpreter. This Interpreter is responsible for transforming the data in the internal data representation of a tool to ISyCL, and vice versa.

To achieve effective knowledge sharing, the object-based repository is coupled with the functionality to support change management. Change management is (1) the enforcement of rules and constraints to ensure consistency in the knowledge base by detecting conflicting pieces of information and (2) the automatic propagation of rules and constraints to deduce new information. Through the use of a Constraint Propagation System (CPS), information asserted to the object repository is not only stored in the repository but is interpreted to determine if the information provided contradicts information that had been previously asserted.

Providing change management support does not solve the entire information integration problem. Information integration is also hindered by the difficulty of interpreting data. The Level 3 IDSE provides data interpretation support through the capture and reuse of method and domain ontologies — repositories that capture

---

[6] The IDSE was originally conceived as four distinct architectures labeled Levels Ø, 1, 2, and 3 IDSE, respectively. Though Levels 2 and 3 were originally designed separately, at implementation the architectures were deemed similar enough to warrant a single implementation. Because this single implementation subsumes the functionality of both the Level 2 and Level 3 IDSEs, this implementation is actually the Level 3 IDSE.

the relevant background knowledge and vocabulary of a given domain together with the constraints that link that knowledge to other ontologies. In addition to serving as an interpretation tool for understanding data generated by other project members and their implications on other parts of the project, ontologies are used in the IDSE as background knowledge to identify constraints and rules to be enforced in the knowledge base.

An ontology, simply stated, is the theory of "what is." That is, it is a knowledge base containing information about the concepts and relationships that exist between the concepts of a domain. There are two types of ontologies in the Level 3 IDSE: method ontologies and domain ontologies. A method ontology is a meta-level description of the objects, relationships, and constraints pertaining to a particular method. For example, an IDEF1 method ontology would contain objects such as Entity Classes, Attribute Classes, and Link Classes, and would also capture IDEF1 constraints like the no-repeat and no-null rules. A domain ontology is a meta-level description of the universe of discourse relating to a specific domain. For example, if automobiles are being produced, the domain ontology might include such objects as Engine Blocks, Transmissions, Steering Wheels, and descriptions of the relationships between them. The insights gained from the Ontology Thrust facilitated the development of the Level 3 IDSE ontologies.

Method and domain ontologies used together guide the interpretation of information within the Level 3 IDSE repository. During assertion of information, the data is compared with the statements and rules provided by the method and domain ontologies to determine if new knowledge can be derived. If so, this new information is asserted and added to the repository; existing facts and assertions relating to that information are examined to ensure consistency.

Goals

The ultimate goal of the IDSE development effort was to produce an integrated system development environment. That is, the objective was to develop an IDSE system that would (1) be expandable in order to accept new or improved tools as they become available, (2) provide data integration between tools, (3) be customizable to reflect and provide automatic enforcement of the site and project-specific system development process, (4) provide a Artifact Repository in which all life-cycle artifacts associated with a project will be stored, (5) provide automated version control and configuration management for all life-cycle artifacts, and (6) contain a full range of project development and management tools.

Accomplishments

The Experimental Tools Thrust has focused on Levels Ø, 1, and 3 IDSE experimentation and development. Demonstration systems of the three architectures have been developed. The following sections provide an overview of the work accomplished and highlight some of the more interesting results.

Level Ø Integrated Design Support Environment Experimentation

During the Level Ø IDSE experimentation, most of the development centered on artifact management and capabilities related to artifact management. The following subsections describe the Level Ø IDSE.

Artifact Management. Artifact management functionality represents one of the underlying capabilities necessary for an information-integrated environment. The artifact repository provided by the Artifact Manager is a library for system data; it gives an organization a repository of information and knowledge about a system under development. The advantage of an on-line repository is that the data is made accessible to development team members and to integration mechanisms built as part of the IDSE. As a result, an organization benefits by making information easily accessible to team members. In addition, integration can be automated as this data is made available to the IDSE for automated exchange of data and information.

An *artifact*, from the Level Ø IDSE view, is any of the various reports, models, business policies, products, and so forth used by an enterprise. As mentioned, the Artifact Manager serves as a library for these artifacts. In keeping with that analogy, an artifact must be checked out before a user can access it and must be checked in before it is made available to other users. By implementing such a procedure, an organization maintains tracking and access control over the artifact, ensuring that only authorized personnel will be allowed to make changes. In addition to functionality for supporting the checking in or out of artifacts, the Level Ø IDSE provides functionality for defining exactly what artifacts will reside in the repository.

With regard to this functionality, four kinds of users were defined to play a role in managing these artifacts:

1. Project Members: Project Members are the personnel in the enterprise who work with the artifact on a day-to-day basis.

2. Project Managers: Project Members are the personnel charged with managing the projects carried out by the enterprise. They define what artifacts will be managed in their projects and determine which users will be Project Members of their projects.

3. Librarian: The Librarian(s) at the enterprise is responsible for carrying out the check-in policies associated with particular artifacts.

4. System Administrator: The System Administrator is charged with defining information about users and projects in the Level Ø IDSE Experimentation System and with defining new versioning schemes for use by the system.

Each of these user types has a set of supporting functions. For Project Members, the Level Ø IDSE provides functions for browsing the artifacts in the system, checking artifacts in and out of the system, and translating artifacts. For Project Managers, the Level Ø IDSE provides functions for adding user roles to their

projects, creating new artifact types with an Artifact Template Editor, and creating artifacts from templates (e.g., the first version of a technical report from a report template). For the Librarian, the Level Ø IDSE provides functions for starting the check-in policy for a specific artifact and for updating the status of ongoing check-in processes. Finally, for the System Administrator, the Level Ø IDSE provides functions for creating and modifying user records, projects records, and other items stored in the Level Ø IDSE's namespace; viewing and creating version schemes for use in the system; and maintaining the persistent artifact database.

Versioning Support. Part of any artifact management system is its versioning capability. Versioning allows the artifact manager to maintain a history of the artifact evolution and to support alternate development paths through version branches. The Level Ø IDSE implements the ORION version model developed by Microelectronics and Computer Technology Corporation (MCC) (Banerjee, Chou, Garza, & Kim, 1986).

However, an objective of the IDSE is to allow the environment to adapt to the policies and practices of the organization in which the IDSE would be installed. Because different organizations have different strategies for maintaining versions of artifacts, the Level Ø IDSE provides an organization the ability to define its own version model. This ability was accomplished by identifying a set of basic operations on versionable objects to be supported by the versioning system. The four basic operations defined for a versionable object are summarized below.

Delete Version: The *delete version* operation removes an artifact from the version derivation tree. Although the artifact is deleted, the fact that it once existed is captured in the derivation tree so that artifact evolution can still be tracked.

Derive Version: The *derive version* operation derives a new version from an existing one. In some cases, the type of operations (e.g., derive, promote, delete) that can be performed on the existing version will change when the new object is derived.

Promote Version: The promote version operation causes a version to be promoted (i.e., to undergo a state change).

Replace Version: The *replace version* operation allows an existing version of an artifact to be replaced with a new artifact.

When users define a version model, they specify the various states in which a versioned object may exist, what operations from the previous list may be performed on objects in a certain state, and to what state the object transitions when a basic operation is performed on that object. Figure 3 provides an example to assist in properly describing this concept. This figure shows the version model derived for the ORION versioning scheme. In this scheme, objects can exist in three states: working, transient, or inaccessible. The figure shows the various

transitions that objects in each of these states can undergo, and also specifies if new objects should be created by the operation and the states in which those objects should be created.

Operation	Beginning State	Ending State	New Object State
Derive	Working	Working	Transient
	Transient	Working	Transient
Promote	Transient	Working	N/A
Replace	N/A	N/A	N/A
Delete	Working	Inaccessible	N/A
	Transient	Inaccessible	N/A

Figure 3

ORION Version Model Representation

In the ORION model, new transient versions can be derived from working versions. As a result, ORION allows multiple transient versions to be derived from the same working version. Thus, ORION allows a true version tree because multiple derivation paths can be generated from the same working version. However, if an organization only wanted to allow a linear derivation path with no branches, a new version model could be defined. To do this, the ability to derive a new transient version from a working version would be removed from the state transition table shown in Figure 3.

Since many different version models can be defined in this manner, the Artifact Manager requires that the version model be specified in the definition of the artifact. This definition ensures that the Artifact Manager will enforce the proper version derivation rules. Furthermore, an artifact could be defined as not being a versionable object at all. In this case, the artifact is managed as a single object with no related version derivation history maintained.

Automated Check-In Policies. Perhaps one of the most exciting developments of the Level Ø IDSE experimentation involves the automation of artifact check-in policies. A design, report, model, or other important document often undergoes a formal review process before it is finalized. In providing artifact management, it is important to support and automate, as much as possible, this review process. The advantage of an automated review process is that the IDSE ensures that the review process is followed, monitors the progress, and tracks the results of the review process. This review history becomes an important part of the evolution of the artifact.

Using an IDEF3 description of an artifact review process, the Level Ø IDSE automates much of the review process. This automation is accomplished using an activation mechanism for IDEF3 descriptions. Essentially, this activation mechanism simulates the process defined in the IDEF3 description. Through this simulation, the IDSE is capable of indicating to the Librarian (i.e., person responsible for ensuring the review process is followed) what

actions need to be taken for an artifact to undergo a proper review. With the more sophisticated levels of the IDSE, many actions specified in the review policy will be automated, thus reducing the importance of the Librarian's role.

Like version models, many different review policies can be maintained by the Artifact Manager. Also, the Artifact Manager requires that the review policy be attached to the artifact definition so the policy can be enforced during artifact check-in. If no review is required for an artifact, the artifact can be defaulted to having no review policy attached to it.

The review policy functionality is essentially the same functionality required for framework processing to be introduced in the Level 1 IDSE. A framework is a representation of the system development process of an organization. This effort has experimented with frameworks at Level 1, and with monitoring, controlling, and managing the system development process automatically. In formulating this framework processing concept, IDEF3 was identified as a good representation for these frameworks. By developing functionality for processing relatively small review processes, work has begun towards providing full-scale framework processing capability at Level 1. This framework processing capability is discussed further in the Process Layer subsection of the next section.

The review policy functionality of the Level Ø IDSE demonstrates the kind of automated functionality that can be provided by a computer system when information (in this case, the review policy) is structured and made available to the system. Artifact Translation is another example where the Level Ø IDSE applies information maintained about individual artifacts to automate user tasks. Within the Level Ø IDSE, the actual artifacts are maintained in their original format. However, it is very common for data present in one artifact to be needed or referenced in another artifact in another format. For this reason, the Level Ø IDSE established functionality to automatically execute artifact translators. With this functionality, the user would select the desired artifact and the desired format for that artifact. The Level Ø IDSE would then access the information maintained about the artifact (mainly its original data format), determine if an appropriate translator from the source format to the destination format was present, and then execute the translation. The Level Ø IDSE required that custom translators be built, but if the translators were constructed and made available, the Level Ø IDSE would transparently execute the translator and make the new artifact available to the user. Much like the automated review policy was a precursor to the Level 1 IDSE's Framework Process, this automated translation capability was the precursor to the Level 1 IDSE's Integration Services to be discussed in the following section.

Collectively, the Automated Review Policies and the Artifact Translation capabilities of the Level Ø IDSE represent the most significant accomplishments of Level Ø. Besides serving as the inspiration and basis for more sophisticated capabilities built into the Level 1 IDSE, the Review Policy and Artifact Translation capabilities demonstrate how information about business processes and artifacts, when made available to computer systems, can be applied to automate and support an organization's business systems.

Level 1: Integrated Development Support Environment Experimentation

During the Level 1 IDSE experimentation, development focused on several areas, including enhanced data and function integration, development process management, and distributed data management and networking. Figure 4 presents the conceptual architecture of the Level 1 IDSE. As shown in the figure, the architecture adopts a layered approach.

```
Process Layer
Integration Layer
Communication Layer
```
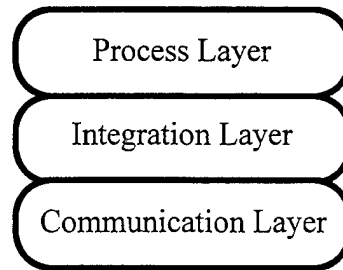
Figure 4
IDSE Conceptual Architecture

This architecture is grounded on the philosophy that each layer enhances functionality by organizing and structuring the layer below it. In the IDSE, the communication layer ties various computer systems together and allows communication between those systems. The integration layer then provides protocols and services that allow software systems and data management systems residing on those different computer systems to communicate and cooperate. Finally, the process layer allows product development team members to communicate and cooperate better. Support for CE is affected through the process layer by effectively sharing data between, and coordinating the activities of, the product/project team. The next three subsections address each layer of the Level 1 IDSE architecture.

The Process Layer. The process layer, also known as the framework processor, captures descriptions of processes and uses them to manage product evolution. This capability surpasses traditional integration efforts where the focus of the environments has been on the integration of tools and data. Because integration of software tools and data across computer systems does not guarantee successful application of CE, the IDSE has added this process-oriented capability.

The approach for incorporating process knowledge into the IDSE evolved from John Zachman's research into frameworks for information systems architecture (Zachman, 1986). Figure 5 displays Zachman's original framework. In this context, a *framework* is an architectural representation that defines the boundaries and situations in which methods and tools are appropriate to the development of a system. The framework is a matrix with the rows representing the various perspectives or viewpoints in an organization. These rows organize the various descriptions of the system architecture with respect to the multiple viewpoints on the system (e.g., the executive [Objectives], the manager [Model of the Business], the programmer [Detailed Representation], etc.). The columns

of the framework represent specific focuses of the system descriptions. The junction of a row and column (i.e., a cell) represents a situation with a particular focus from a specific user viewpoint. Each of these different situations can be described and defined using different tools, methods, and techniques. Consequently, a collection of all the descriptions produced in each of these situations represents a complete description of the information system architecture.

	DATA	FUNCTION	NETWORK
OBJECTIVES/ SCOPE	List of things important to the business ENTITY = Class of Business Thing	List of processes the business performs Process = Class of Business Activity	List of locations in which the business operates
MODEL OF THE BUSINESS	e.g., Entity / Relation Diagram ENTITY = Info. Entity RELN = Bus. Rule	e.g., Function Flow Diagram	e.g., Logistics Network Node = Bus. Unit Link = Bus. Relatn
MODEL OF THE INFORMATION SYSTEM	e.g., Data Model ENT = Data Entity RELN = Data Reln	e.g., Data Flow Diagram	e.g., Distributed System Arch. Node = Information System Function Link = Line Char.
TECHNOLOGY MODEL	e.g., Data Design ENT = Segment RELN = Pointer	e.g., Structure Chart	e.g., System Arch. Node = Hardware Link = Line Spec.
DETAILED REPRESEN- TATIONS	e.g., Data Design Description ENT = Field RELN = Address	e.g., Program	e.g., Network Architecture
FUNCTIONING SYSTEM	e.g., Data	e.g., Function	e.g., Communication

Figure 5

Zachman Framework

Zachman's framework attempts to capture both the different situations that exist during the life cycle of an information system and the type of information that describes the information system in that particular situation. Figure 6 shows a derivative of the Zachman framework representing a single cell of the Zachman framework. A framework of this type would be defined for each cell of the Zachman framework. The cell definition would capture the activities to be performed, the role of the personnel participating in those activities, and the information that must be discovered, decided upon, or managed.

In essence, this framework captures a complete description of the product development process. By imposing a structure on this framework information, the Level 1 IDSE can be "programmed" to control and manage

39

a product development process — the fundamental idea behind the process layer. Given a structured description of the product development process within an organization, the process layer coordinates and tracks the development activities. Therefore, the components of the process layer focus on the capture of development process descriptions and on the ability to interpret those descriptions. Within this layer, the Level 1 IDSE uses the IDEF3 Process Description Capture Method to capture and structure the information necessary to manage, coordinate, and monitor the product life cycle. (The *IDEF3 Process Description Capture Method Report* [Mayer et al., 1992a and 1995d] provides more information.) Using these IDEF3 descriptions, an enterprise can capture the best practices for a given development process as well as incorporate CE strategies into its development process.
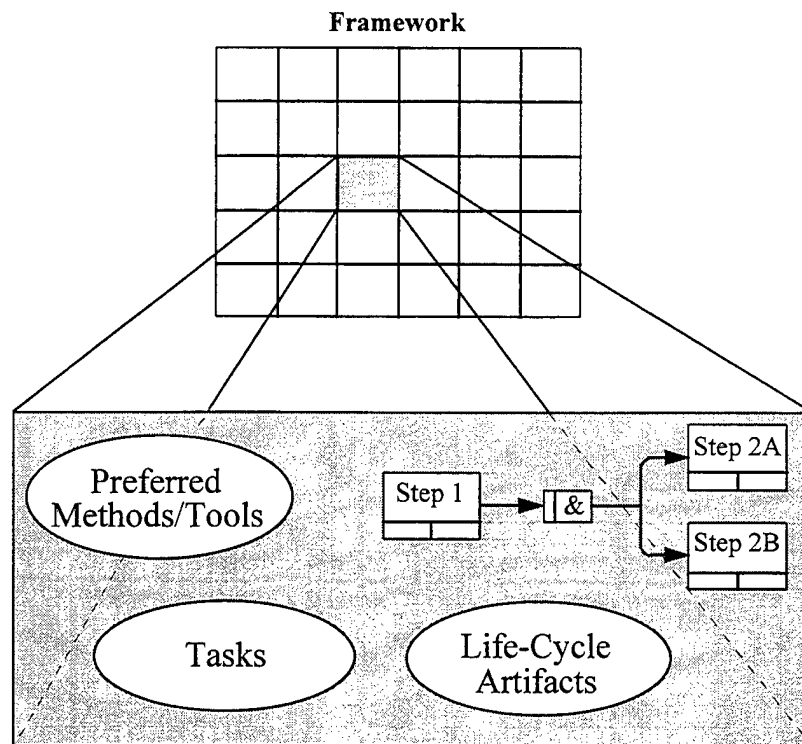
Figure 6

Framework Cell Contents

The process layer consists of three primary components: process manager, process repository, and process parser (Figure 7). The process manager monitors and displays the state of the development process. Users have access to the process information through the graphical syntax of the IDEF3 process language, coupled with the development status information maintained in the process repository. By browsing this process description, augmented with process status information, users can determine what development tasks are currently active and where they should focus their efforts. Based on this information, users can make requests to the process manager to have certain tools invoked or to gain access to life-cycle artifacts pertinent to the product development.
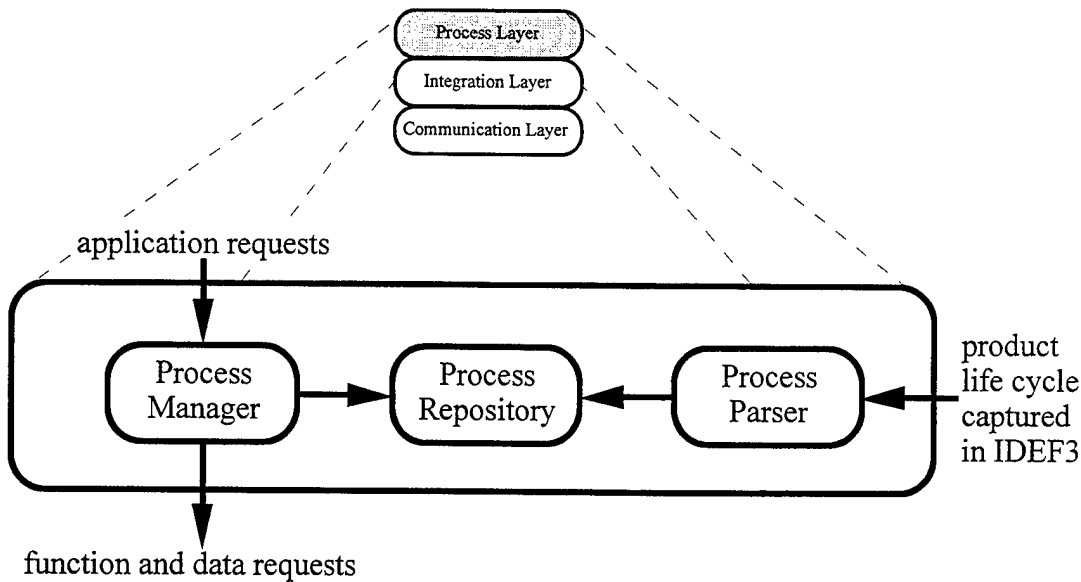
Figure 7

Process Layer

The process manager manages, updates, and monitors multiple development processes in the Level 1 IDSE. All messages to the Level 1 IDSE are filtered through the process manager, which determines whether to allow the given message based on the access control privileges of the requester. After authenticating the message, the process manager routes the message to the integration layer for processing. In addition, the process manager determines whether the message indicates an event that affects any of the development processes it is managing. If so, the appropriate development process is updated to reflect the new status based on the message.

The other two components — the process repository and the process parser — support the process manager. The process repository maintains the set of process descriptions contained in the CE environment and provides a query language to allow applications to access the information maintained about a development process. The process parser reads an IDEF3 based product life-cycle description and performs syntactic and semantic checks to ensure the integrity of the information. This component operates like a language compiler. Upon successful completion of these checks, the process description is given to the process repository for storage.

The Integration Layer. The integration layer is beneath the process layer. A considerable portion of any CE environment is the information systems used to support the concurrency in the process. The purpose of the integration layer is to leverage existing tools and systems better. In typical engineering environments today, various groups tend to have their own task-specific computer systems. Providing function and data integration of these different systems will improve the communication and effectiveness of product development team members.

41

In addition to integrating tools used by the development team, the CE environment must do the following:

1.   Manage and propagate constraints on evolving product-definition artifacts.

2.   Transparently control the configuration of the artifacts.

3.   Support rapid browsing and modification of information about an evolving product.

4.   Support definition and management of versioned objects.

5.   Provide configuration decision management capabilities.

Without these capabilities, product development can be delayed and quality diminished. Lack of computer support for determining the effect of changes can result in major delays in a product program. Inability to enforce constraints across multiple representations of design data often results in a requirement for the design data. Normally, configuration control is performed manually, causing delays in locating current design data.

These points highlight the need for a computer environment that combines a flexible integration strategy with underlying artifact management capabilities to support a CE environment. To satisfy these needs, the integration layer can be viewed conceptually as consisting of two primary components: the integration services manager and the data services manager (Figure 8). The rationale and operation of these two components are discussed in the following sections.
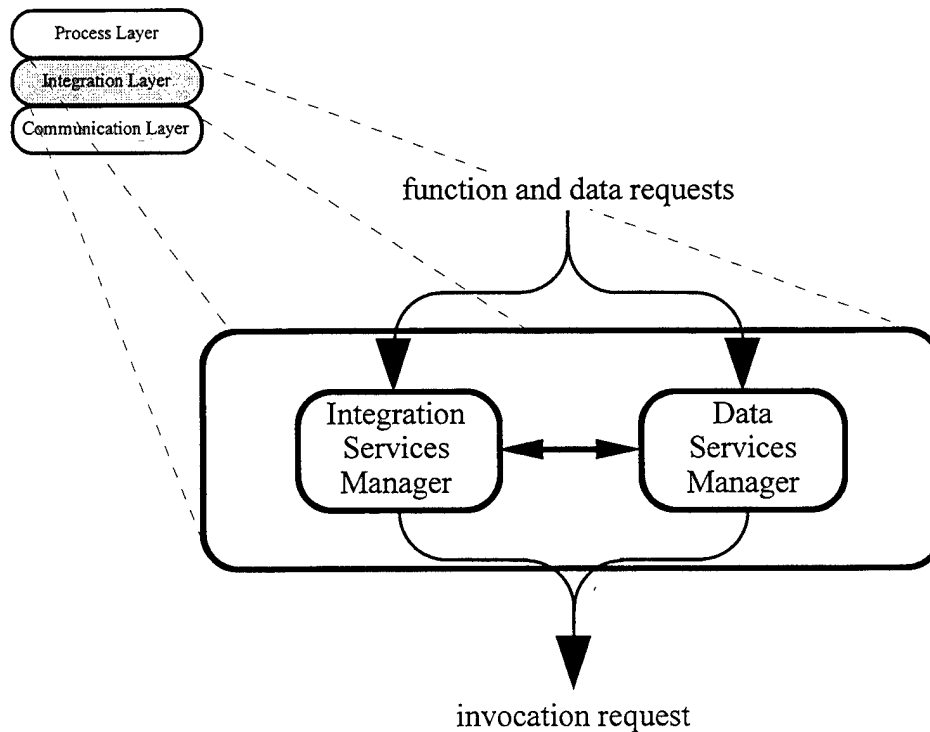
Figure 8

Integration Layer

42

The Integration Services Manager. The integration services manager is based on the theory that the integration of an engineering application into a CE platform should provide greater functionality to the system by adding new services and resources. The *integration services* approach represents a new way of looking at the integration problem. In essence, the focus changes from constructing an "integrated system" to developing the infrastructure to better leverage and integrate the services already provided by various applications.

In previous approaches to integrated systems architecture, the burden to provide integration support has been on the platform [(Linn & Winner, 1986) (Rockwell International, 1989)]. This traditional approach has severe problems. One problem involves the need to define, in advance, comprehensive standards, and to build applications that meet these standards. This presumes an organization can foresee the integration services that will be required and the relative demands for those integration services. Presuming an organization *can* foresee the services required and that the relative demands for those services are unknown, traditional integration approaches force a leveling of integration support across all needs. This implies a massive overhaul of existing legacy systems and unjustifiable modifications by vendors of their existing systems to achieve even a minimal level of integration support. In other words, traditional integration efforts have focused on large, complex environments that incorporate all the functionality required by the organization, regardless of the relative demands for that functionality.

An approach is needed where services can be introduced incrementally as "user demand" forces the needed services to emerge. Appropriate guidelines and structures for representing and executing these services must be established. Once the expense of setting up a service has been incurred, that service should be available to all subscribers and other services will evolve in an organized fashion. With a service-based approach, an application advertises the services it provides as well as the invocation procedures for that service. The advertisements define external interfaces that allow other tools to take advantage of the functionality provided by the new application. The integration layer organizes and maintains these interface definitions and routes the integration service requests.

This flexibility in the integration mechanism is important when considering the overall CE process. Previous integration efforts have focused on a particular domain and ways in which elements in that domain interact (Linn & Winner, 1989; CAD Framework Initiative, Inc., 1991). This vertical integration, while important and useful, does not address how the various domains (e.g., marketing, manufacturing, and design) interact. A problem with integrating systems from these different areas is that often it is not known when and where tools should be integrated. However, with the integration service approach, services supporting known interactions can be provided initially; new services can then be developed as the demand arises. The flexibility of the service approach allows support for these processes to be added as the processes are learned and developed.

Using this service approach to integration, the integration services manager can be viewed as an information booth of the services available in the Level 1 IDSE. Because the Level 1 IDSE operates in a distributed

heterogeneous computing environment, the services may reside on the local host or on a remote host. The integration services manager, not the requesting agent, determines where and how to invoke the service.

The integration services manager can be conceptually decomposed into four functional units (Figure 9). The service registration tool allows the addition, modification, and deletion of services supported by the Level 1 IDSE. The service registration tool collects and maintains the following information about a service: where the service is located, what the service does, and how to invoke and control the execution of the service. All this information must be structured to allow the integration services manager to query the requester for the information necessary to invoke the service.
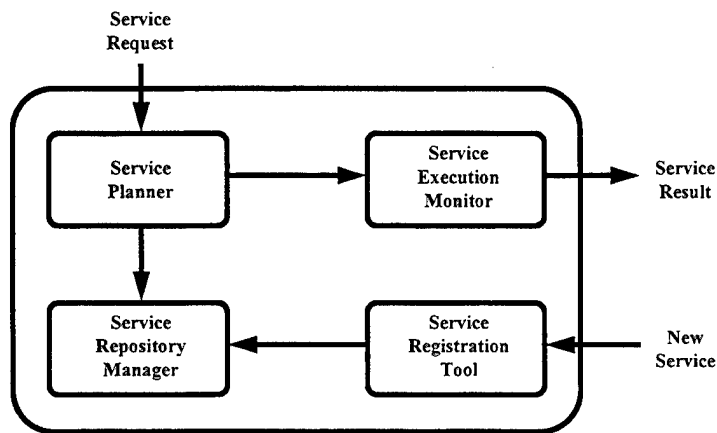
Figure 9

Integration Services Manager

The service registration tool presents this information to the service repository manager, which maintains the knowledge base of services and provides a query language for other applications and the process layer to request service information. The service repository manager also maintains the status of the computer systems on the network and uses this information to inform a service requester of the availability of a given service.

The service planner receives all service requests and examines the service repository to determine whether the request can be satisfied by an existing service or service plan. A *service plan* is a sequence of services that collectively satisfy a given service request. A *step* in the service plan represents an individual service invocation. If the request cannot be satisfied directly, the service planner attempts to develop a service plan to accomplish the given service request. A service request can be satisfied in three ways by the service planner: (1) by a registered service, (2) by a preexisting service plan, or (3) by a newly generated plan developed by constructing a sequence of services that satisfy the request.

Once a service plan for a request has been developed, the service planner passes the plan to the service execution monitor. This component queries the requester for additional information necessary to invoke any service

44

in the service plan. Having acquired all necessary information, the service execution monitor begins sequentially invoking each step in the service plan. If the service resides on the local host, the invocation structure for the service is constructed from the information contained in the service plan and a process is created to execute the service. However, if the service resides on a remote host, the service step is routed through the communication layer to the appropriate host with the result being returned to the local host.

The Data Services Manager. Using only the integration service approach to link the various tools used during product development will neither improve the quality nor reduce the development time of new products. This integration capability must be coupled with the ability to manage and control access to data produced by the various tools and systems used during product definition. After all, the *data* produced with integrated tools are important, not the fact that the tools are integrated. The data services manager creates, manipulates, accesses, and stores *data* while the integration services manager provides similar support for *functions*.

The data services manager manages life-cycle artifacts. An *artifact* is an electronic data item that maintains information about an object that is created or manipulated during the development process. For example, in the software development process a software design document is created during software development. In the IDSE, information such as creator, owner, last modification date, and access privileges are some elements of information maintained in the electronic artifact for the design document. Additionally, the word processor files that compose the design document would be managed as part of the artifact. Thus, objects that participate in a development process are represented by electronic artifacts. Because this information is available electronically, the artifact data can be used to control, track, and monitor the object during its life cycle.

In managing life-cycle artifacts, the data services manager registers data artifacts, maintains access control over the artifacts, and provides versioning and configuration management for the artifacts. This type of functionality acknowledges that the artifacts managed by the data services manager are valuable to the organization. Accordingly, the data services manager enforces access control policies established by the organization. These policies specify which users have access to the artifacts managed by the data services. As a result, every artifact operation request must be passed through an authorization procedure to ensure the user submitting the artifact operation is authorized to complete the operation.

When user authorization is approved, conditions that may prevent the execution of the operation are analyzed. For the most part, these conditions involve a certain artifact being in a state that does not allow the requested operation to be performed. For example, if a user requests to check-out a specific artifact, the artifact manager must determine if another user has already checked out the artifact, in which case the artifact is locked and no other user can access the artifact (except on a "read only" basis). If no conditions prevent the execution of the requested operation, the data services manager executes the necessary steps to perform the operation. In the case of a check-out operation, the artifact information is updated to reflect that the artifact has been checked out; a message

45

is then sent to the user indicating where the artifact is located. For a check-in operation, the artifact information is updated and the artifact manager copies the artifact into the artifact library.

In performing these functions, the data services manager uses the integration services manager to invoke and execute many of the functional services required to support artifact management. The data services manager takes advantage of the functionality provided by the integration mechanism to store and manipulate artifact data. Using this approach, artifact information does not necessarily have to reside on a single machine. Because of this interaction with the integration services manager, the data services manager is capable of providing a heterogeneous, distributed artifact repository.

The Communication Layer. To operate in a distributed environment, the integration layer depends on the communication layer — components providing a high-level message-passing capability in a heterogeneous computing environment. The conceptual decomposition of the communication layer is shown in Figure 10.
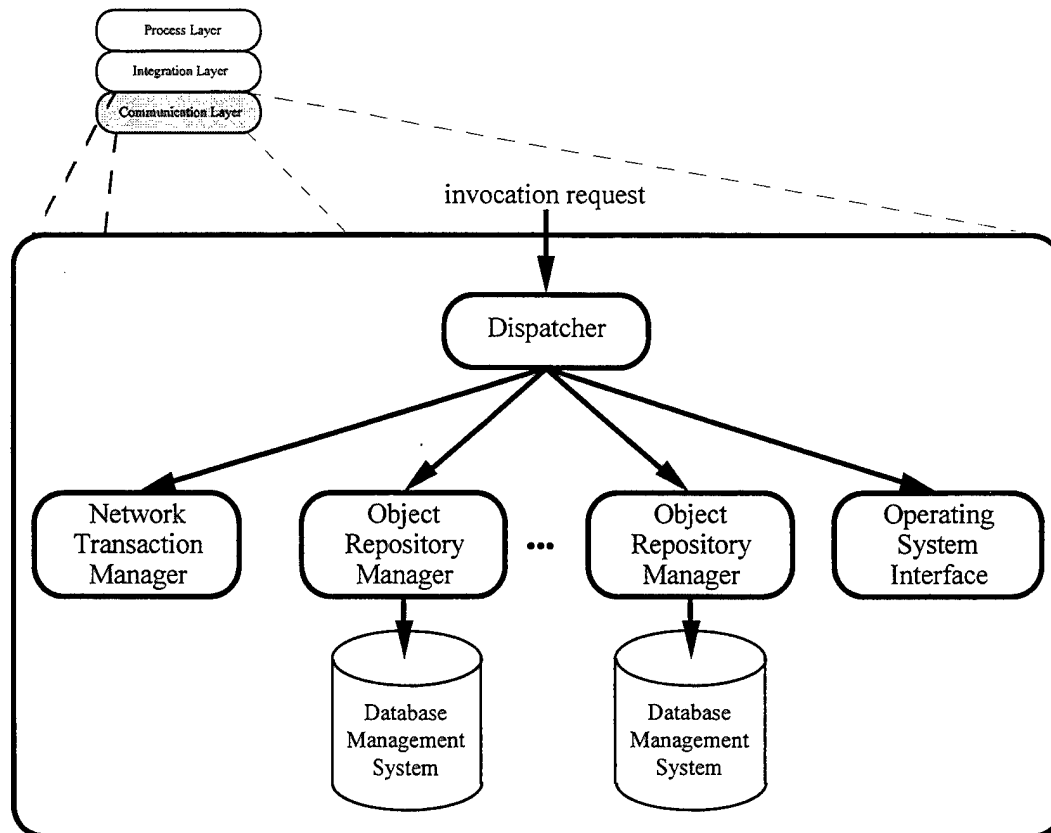
Figure 10

Communication Layer

To cope with the distributed nature of data and services, the dispatcher routes requests among the computer systems in the environment. Data requests may involve local or remote object repository managers. However, the data request is independent of the location of the object repository manager. Instead, the dispatcher processes the

46

generic data request and determines where the data reside in the environment. If the requested data reside locally, the dispatcher routes the request to the appropriate local object repository manager. Remote data requests are given to the network transaction manager, who parcels the request to the remote host for processing.

The object repository manager converts requests from the IDSE representation to the native representation used by the database management system. The native request is given to the database management system, and the results are converted from the native representation back to the IDSE representation. Thus, the object repository managers are intelligent interfaces between a database management system and the IDSE.

A service request spawns a local or remote process which performs the request. The dispatcher determines the computer system on which the process will be spawned to provide the service. Local service requests are passed from the dispatcher to the operating system interface. This interface invokes local services and returns the results to the dispatcher. Just like the remote data requests, the remote service requests are given to the network transaction manager to deliver to the remote host for invocation.

As an extension of the Automated Review Policies and Artifact Translation of Level Ø, the Framework Processor and Integration Services Manager represent the most significant accomplishments of the Level 1 IDSE. The Framework Processor not only makes business processes accessible and available to users but monitors, coordinates, and facilitates the execution of those processes. The Integration Services Manager not only enables the Framework Processor but enhances the application of computer resources by making the computer resources more valuable to the organization.

Level 3: Integrated Design Support Environment Experimentation

Experimentation with the Level 3 IDSE represents a break from the focus of both the Level Ø and Level 1 IDSEs. Levels Ø and 1 focused on technology to support the coordination of activities by organizing and managing project artifacts, as well as monitoring and tracking the state of project activities. Level 3 is much more focused on the knowledge gained and shared during a development or analysis effort. The Level 3 IDSE represents an extension of the capabilities provided by Levels Ø and 1 to include a major advance in information integration functionality. The Level 3 IDSE is capable of reasoning about the *semantics* of the data that it manages. This allows the system to provide automatic model translation capabilities[7] and support for data consistency across the entire system development process.

---

[7] Automatic model *translation* should not be confused with automatic model *generation*. The Level 3 IDSE has no way of ensuring that the projection of the information contained in a model in one method forms a complete model in another method. In fact, many of the translations result in partial models.

Level 3 IDSE Approach. The proposed approach to the problem of design knowledge sharing and design information integration is to provide an integrated environment in which information pertinent to a design project is stored in a (conceptually) global repository. The global repository can then be manipulated and managed to support change management, decision rationale capture, and consistency maintenance

The Level 3 IDSE focuses on information sharing and data interpretation. The design of the Level 3 IDSE is based on the conviction that information integration cannot be achieved without support for both change management and data interpretation. Change management is provided by the automatic propagation of rules and constraints to deduce new information and the enforcement of these rules and constraints to ensure consistency in the knowledge base and detect conflicting pieces of information. An example of how such propagation and enforcement support change management is given later in this chapter.

The other mechanism in the IDSE that supports information integration is data interpretation. Information integration is not only hindered by inaccessible data and a lack of change management support but also by the difficulty of interpreting data. The IDSE provides data interpretation support through the capture and reuse of domain ontologies — repositories that capture the relevant background knowledge and vocabulary of a given domain, together with constraints that link that knowledge to other ontologies. Ontologies are used in the IDSE as background knowledge to identify constraints and rules to be enforced in the knowledge base, as well as an interpretation tool for understanding data generated by other project members and their implications on other parts of the project.

The concept of operations for the Level 3 IDSE is as follows. Members of a design project team record and store information about the design or part of the design of a new product using their favorite software tools which have been integrated with the Level 3 IDSE. When their part of the design is relatively stable, they assert the information pertinent to their part of the design into the global repository. These assertions may trigger changes in the repository or may cause a conflict in the knowledge base. If a conflict is detected, the assertions that cause the conflict are automatically retracted, and the user from which the assertions originated is informed of the situation. If the assertions caused changes in the knowledge base, the originator is also informed of these changes. In general, changes to the knowledge base are propagated using constraints and/or rules that have been defined on the information stored in the knowledge base. Later, project members can retrieve the part of the design they are working on from the global repository and load it in their preferred tool. All changes in the knowledge base that have affected their part of the design will be apparent in the information they retrieve. Users can also access and manipulate information in the knowledge base directly through a "Browser."

The functionality of the Level 3 IDSE can be illustrated easily with the help of an example. Consider a project dealing with the development of a computer software for a Department of Defense (DoD) agency. The

development of the software must follow the DoD standard for software development (2167A). A possible use of the Level 3 IDSE is as follows.

1. The contractor has developed a function model (IDEFØ) of the activities that need to be performed during the software development effort and has stored the model in the global repository. (This model is referred to as the "general model.")

2. The project manager uses a software tool registered with the IDSE, such as AIØ Win, to retrieve the general model from the knowledge base, then modifies the model to obtain a specific function model for the current project and asserts that new model in the knowledge base. (This model is referred to as the "specific model.")

3. The project manager decides to keep some links between the general model and the specific model so that changes to the general model are reflected in the specific one. Therefore, using the browser, the project manager asserts some constraints and rules in the knowledge base to record dependencies between the two models.

4. A new standard for software development is published soon after, and an employee of the contracting company is assigned the task of modifying the general model accordingly. After retrieving the model from the global repository, the employee modifies it according to the new standard and re-asserts it to the knowledge base using any available function modeling tool.

5. The reasoning component of the Level 3 IDSE detects a change in the general model and executes or triggers the rules that were entered in Step 3. One rule causes a message to be sent to the employee who modified the general model. The message informs the employee of the impact of the changes on the specific model. Other rules cause some elements of the specific model to be modified.

6. The employee informs the project manager of the messages sent by the reasoning component regarding the specific model.

7. The project managers retrieves the specific model from the knowledge base to assess the changes and their impact on the current project. In doing so, the manager notices that some of the changes affect a deliverable that is unfamiliar. To ensure the change have been interpreted correctly, the manager browses an ontology of software development deliverables that has been recorded and stored in the knowledge base using the ontology capture and browsing tool (OCBT).

Level 3 IDSE Conceptual Architecture. The conceptual architecture of the Level 3 IDSE is shown in Figure 11. The environment is composed of tools and a core component that consists of a dispatcher, an ISyCL

projector, an OCBT, an evolving system description (ESD), and an ESD browser. A brief description of each component follows.
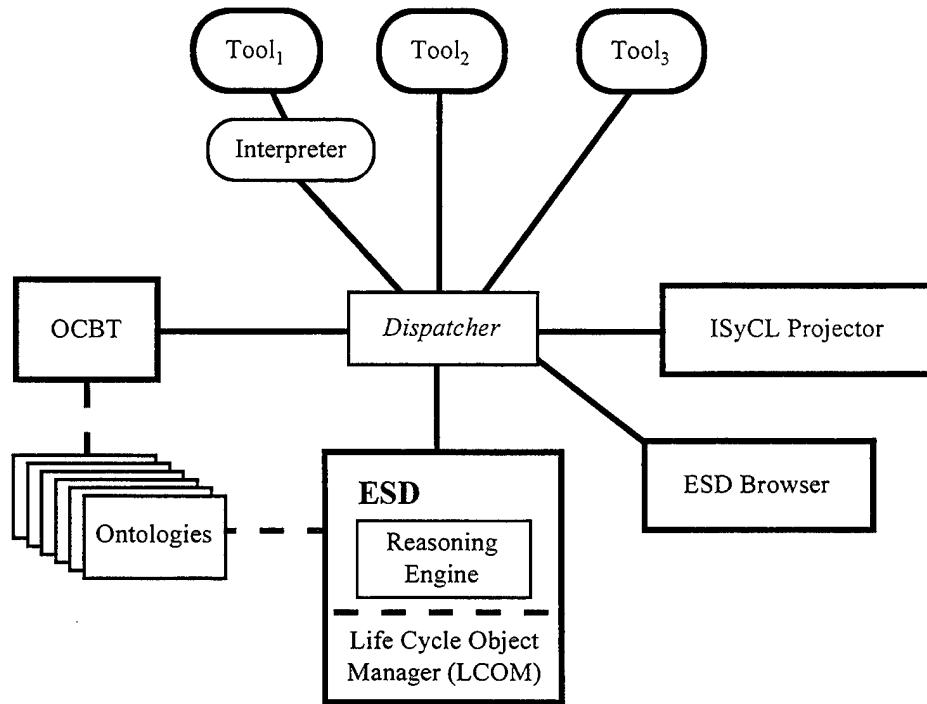
Figure 11

Level 3 IDSE Architecture

- **Tools.** Tools, as represented by $Tool_1$, $Tool_2$, and $Tool_3$ in Figure 11, are software applications that can produce information to be stored in the ESD and/or that need to retrieve information from the ESD. These applications, which must be registered with the environment before they can communicate with the ESD, communicate with the ESD in ISyCL. Tools that cannot generate ISyCL statements must use an application called an *ISyCL interpreter*.

- **ISyCL Interpreter.** An ISyCL Interpreter enables applications to communicate with the ESD. It enables applications to send information to the ESD by producing a set of ISyCL statements that represent information generated by the application and to retrieve information from the ESD by interpreting ISyCL statements and generating corresponding data in a format adequate for the application. Each application that is part of the environment has an associated ISyCL Interpreter.

- **OCBT.** The OCBT enables the capture and management of domain and method ontologies. It provides for the automatic registration of applications with the IDSE, the creation of object structures in the ESD based on ontological descriptions, the assertion of dependencies between ontological elements, and the assertion and retrieval of domain and method ontologies into and from the ESD.

- **ISyCL Projector.** The ISyCL projector is both the mechanism by which applications are registered with the IDSE and the mechanism by which ISyCL statements coming from ISyCL interpreters or the OCBT are interpreted. It provides functionality for parsing application and method ontologies, and for creating and manipulating ESD objects. It also supports the generation of ISyCL statements to be sent to applications in response to a "retrieve information" request from an application.

- **ESD.** The ESD consists of a knowledge repository to store information provided by tools and a reasoning engine to support change propagation and consistency maintenance. Object management in the ESD is provided by the life cycle object manager (LCOM). The LCOM provides facilities for creating, updating, and deleting objects in the repository and for defining relations among those objects. It includes components for artifact configuration management and version control. The ESD's reasoning engine is the constraint propagation system (CPS) that supports the enforcement of constraints and rules on objects in the ESD or their attributes. The CPS also supports the enforcement of universally quantified constraints.

- **ESD Browser.** The ESD browser enables the direct manipulation of information stores in the knowledge repository. It provides for the creation and deletion of objects and schema, for the modification of schema and objects (such as the addition or deletion of an attribute or the modification of an attribute value), and for the assertion of rules and constraints in the CPS.

- **Dispatcher.** All IDSE components communicate through the dispatcher. The dispatcher is responsible for routing messages between the different IDSEs. All commands to the dispatcher are in the Request/Response Language (RRL), a low-level language used by the IDSE components to communicate. The RRL works in a heterogeneous network environment, supports both a command-oriented and an event-oriented communications paradigm, and supports the parallel execution of multiple outstanding messages. It uses a host-independent data format and a peer-to-peer communications paradigm.

A more detailed description of these components and how they interact follows.

Level 3 IDSE Component Description. The following section, which describes the operation of the various components of the Level 3 IDSE, is organized according to the user's perspective of the IDSE. In other words, the elements a user would most likely encounter are described first, followed by a discussion of the "internal" components of the IDSE architecture.

Tools and ISyCL Interpreters. A tool is any software application that provides information to the IDSE. These applications are both the tools that a system development team uses to analyze, design, and develop the

51

system, and the tools that typically provide knowledge in the form of models, design documents, and the like. Before a tool or application can operate in the IDSE environment, the tool must meet two requirements:

(1)   A tool must generate ISyCL statements representing the information the tool manages (i.e., ISyCL statement generation requirement [Fillion, 1995]).[8]

(2)   A tool must be capable of sending messages to and receiving messages from the dispatcher (i.e., communication requirement).

These requirements can be satisfied directly by a tool or by a separate ISyCL interpreter. For a tool to satisfy the requirements directly, it must be modified to include the communication functionality necessary to interact with the dispatcher and must be capable of producing the appropriate ISyCL statements. This functionality could be provided indirectly by incorporating the communication and ISyCL generation functionality into a separate utility. The ISyCL interpreter would take as input the native data formats of the tool, produce the corresponding ISyCL statements, and pass those statements to the dispatcher for delivery to the projector. The use of, or need for, an ISyCL interpreter is determined by the tool developer — the developer can choose to modify a tool or to develop a separate utility (i.e., the ISyCL interpreter). One advantage of ISyCL interpreters is that they can be developed by third parties.

Before a tool's ISyCL statements can be sent to, and accurately recorded by, the ESD, the type of information managed by the tool must be described to the ESD. This description takes the form of an ontology that is captured using the OCBT. Once the ontology has been captured, it is sent to the ISyCL projector for processing using the register tool command of the OCBT. The ISyCL projector parses the ontology and sends service requests to the ESD for the creation (or modification) of objects and object types. These objects and objects types are then used by the ESD to store information coming from the tool. The ontology is also used by the ISyCL projector to determine the tool's *ISyCL slice*. The ISyCL slice for a tool is the set of ISyCL constructs that are valid for the representation of information for a specific tool. Any statements sent by the tool using a construct not in the ISyCL slice for that tool will cause an error in the ISyCL projector.

---

[8] While the IDSE currently limits exchange of information through ISyCL-compliant data, the IDSE architecture allows for relatively easy extension to the modes of communication between the tools and the ESD. This extension is possible through the addition of projectors capable of interpreting statements in a given language. For example, tools that generate SGML statements could be directly integrated with the IDSE, provided an SGML projector is added to the core IDSE. Hence, this requirement could be somewhat relaxed to require that the tools generate statements in a language for which a projector exists.

Once an ontology has been registered, the ISyCL slice generated, and the ISyCL interpreter completed, the tool is successfully registered with the IDSE and can be used to provide information to the ESD. The services that can be provided by a tool or its corresponding interpreter are limited by the services supported by the ESD. Though the ESD provides a number of services for asserting and retrieving information, it is up to a tool or its associated interpreter to decide which services will be implemented at the tool level. For example, two commercial tools are currently registered with the Level 3 IDSE: ProCap-IDSE and AIØ Win-IDSE. Both tools have been modified to generate and read ISyCL statements and to initiate communication with the dispatcher. ProCap-IDSE supports the assertion and retrieval of both models and the individual elements in a model, while AIØ Win-IDSE supports only the assertion and retrieval of models. The ESD is capable of receiving both models and individual elements, but AIØ Win-IDSE implements only the assertion of models.

OCBT. The OCBT enables the capture of ontologies according to the concepts and procedures of the IDEF5 Ontology Description Capture Method (Mayer et al., 1994). Briefly stated, the key concepts of an IDEF5 ontology are: *kind, individual*, and *relation*. Informally, a kind is a group of individuals that share some set of distinguishing characteristics. More formally, kinds are properties expressed by common nouns such as "employee," "machine," and "lathe." Individuals are the most basic kind of real-world objects. Unlike objects of a higher logical order such as properties (including kinds) and relations, individuals are not instantiable. Individuals are similar to "instances" in object-oriented programming, while kinds are similar to "classes." Finally, a relation is an abstract, general association or connection that holds between two or more objects. The OCBT provides facilities to create and describe kinds, individuals, and relations, as well as regroup them to form ontologies. In addition, the OCBT provides support for expressing axioms that constrain the definition of concepts in an ontology.

Within the IDSE, the OCBT plays a dual role: (1) it provides information to the ESD and (2) it serves as a mechanism for registering other tools with the IDSE. In its first role, the OCBT operates like any other tool registered with the IDSE. It generates and reads statements in the OCBT ISyCL slice (the IDEF5 Elaboration Language) and can be used to contribute IDEF5-based system knowledge and information during the evolution of a product or system. In its second role, the OCBT operates as a special component of the IDSE. As mentioned previously, a tool cannot exchange information with the IDSE until it has been registered. The OCBT provides the functionality for this registration process.

The purpose of the tool registration process is to identify, for the ESD, the type of information to be passed to the ESD by a specific tool. During the registration process, when an instance of an IDEF5 element is encountered, a corresponding ESD operation is executed that modifies the schema of the ESD data structures. Once the process has been completed, instances of elements defined in the tool ontology can be created and manipulated by the ESD. For example, if we are registering a tool supporting IDEF3, a kind in the IDEF3 ontology would be "UOB." When the IDEF3 ontology is registered, the statement (i5-kind 'uob) would cause the ESD to create a new

class in its schema called "uob." Once this procedure is complete, the ESD is able to create and manipulate instances of the "uob" class.

This example demonstrates the special role played by IDEF5 in the operation of the IDSE. Mappings between IDEF5 constructs and internal ESD constructs have been defined to enable the extension of the schema of the ESD. These mappings define how the ESD schema should be modified and how relations should be handled when information adhering to the registered tool ontology is passed to the IDSE. Because of the existence of these mappings, the OCBT can serve as the mechanism to register other tools.

The two roles supported by the OCBT are reflected in two IDSE services supported by the OCBT. The *register* operation is used to register a tool with the IDSE, causing the modification and extension of the ESD schema in the manner just described. The *assert* service is used to store in the ESD the information contained in an ontology. The *assert* operation has no effect on the ESD schema; rather, it causes instances of ESD schemas to be created. The only difference between the two operations is what the ISyCL projector does with the ontology. The same ontology could be passed under the *register* and *assert* operations, but different outcomes would result. Notably, the OCBT also supports a *retrieve* operation (the reverse of the *assert* operation) that retrieves ontologies that have been stored in the ESD.

ISyCL Projector. The ISyCL projector is the mechanism by which tools and the ESD communicate. The ISyCL projector serves as the translation mechanism between ISyCL and the internal ESD representation. The ISyCL projector is responsible for (1) parsing statements coming from the tools and updating the ESD accordingly, (2) parsing statements coming from the ESD and relaying the information back to tools, and (3) providing the mechanisms for the automatic registration of tools in the IDSE. The projector's architecture is illustrated in Figure 12.

ISyCL Projector	
Tool Integrator	Statement Parser
Ontology Parser	Statement Builder
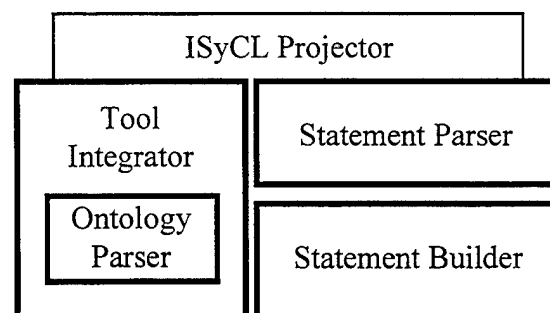
Figure 12

Architecture of the ISyCL Projector

Similar to the OCBT, the ISyCL projector plays a dual role; it is a key component in the tool registration process initiated by the OCBT and also serves as the data interface between the tools and the ESD. The tool

integrator provides functionality to support the tool registration process, while the statement parser and statement builder serve as the data transfer mechanisms between tools and the ESD.

The tool integrator completes the tool registration process initiated by the OCBT by taking a tool ontology as input and performing two operations. The first operation is the modification of the ESD schema through the initiation of ESD operations based on statements extracted from the tool ontology. In this capacity, the tool integrator is the implementation of the mappings defined between IDEF5 structures and the internal ESD structure mentioned previously. The second operation performed by the tool integrator is the automatic generation of an ISyCL slice for the tool being registered. The ISyCL statements produced by the registered tool must adhere to the generated slice because the slice is provided as input to the statement parser and statement builder to ensure proper parsing and generation of ISyCL statements. Though presented in sequence, these two operations are performed simultaneously.

The statement parser and statement builder fulfill the ISyCL projector's second role as the data interface between tools and the ESD. The statement parser is responsible for parsing ISyCL files and individual ISyCL statements sent by tools and for updating the ESD appropriately. It uses the ISyCL slice as the tool to interpret statements and detect syntactic and semantic errors in those statements. Once the statements are validated, the statement parser initiates appropriate ESD services to update the knowledge base. The statement builder performs the analog of the statement parser and is responsible for both interpreting information coming out of the ESD and generating appropriate statements in the ISyCL slice of the tool requesting the data.

ESD. The ESD, which represents the heart of the Level 3 IDSE, is responsible for organizing, storing, and managing information produced during system-development activities. The ESD is composed of two interacting components: (1) a knowledge base called the knowledge repository and (2) a truth maintenance system called the constraint propagation system. The knowledge repository provides for the representation and manipulation of information elements, while the constraint propagation system supports the definition and enforcement of relations and constraints between the information elements.

The knowledge repository provides appropriate representation capability for a wide range of information types at various levels of abstraction. In particular, it supports the representation of complex objects and their components, and the representation of relationships and constraints between those objects. It also supports the representation of different perspectives on an object and the representation of relations between, and constraints on, those perspectives. The knowledge repository representation is based on the container object paradigm (Sanders, Mayer, Browne, & Menzel, 1991). The container object paradigm is a knowledge representation scheme that combines the advantages of the "object-oriented" and the "prototype and delegation" representation paradigms. It differs from traditional representation schemes in two ways. First, it is based on the philosophical idea that an object's existence should be distinguished from the collection of descriptive properties it exemplifies. Hence, in the

55

container object paradigm, an object's existence is independent from its membership in a class. This separation allows objects to be modified dynamically by adding or deleting elements in their descriptions without changing the objects' identities.[9] This approach allows the representation systems based on the container object paradigm to support both complex structured representations of information and the dynamic modification of descriptive knowledge.

The knowledge repository supports the notion of the representation of a "real-world object" having multiple perspectives. The idea is that two different views of an object might be stored in the knowledge repository, and it might be of interest to represent that fact. For example, a function model might contain an activity called *manufacture part*" which corresponds to (or is a perspective on) the real-world activity of manufacturing a part. Similarly, a process model could contain a process called *make part* that is another perspective on the real-world activity of manufacturing a part. If both models are stored in the knowledge repository, the activity from the function model and the process from the process model will be stored in the knowledge repository without any specific link or relationship being represented between the two objects. The notion of a real-world object representation allows that relationship to be captured and represented explicitly.

An example of the implementation of an object having several perspectives attached to it is shown in Figure 13. In this example, the object RWO3 has two perspectives: an activity perspective labeled *MFG-part* and a process perspective labeled *Make-part*. The activity perspective is represented by an instance of the schema (or type) *activity*, while the process perspective is represented by an instance of the schema *process*. Both perspectives are directly accessible from the representation of the real-world object.
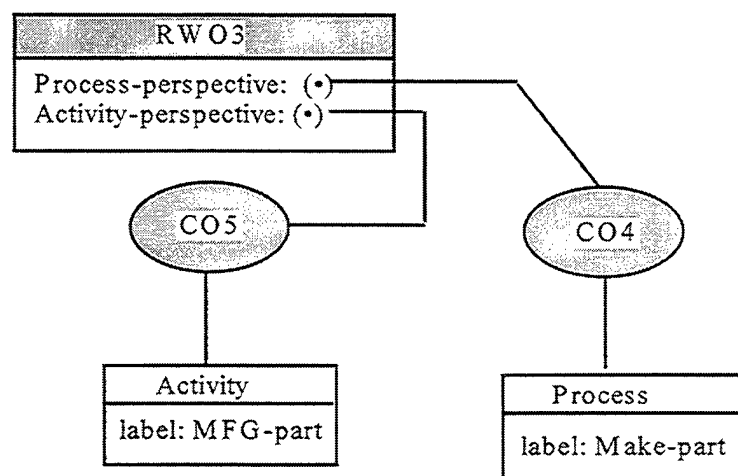
Figure 13

Implementation of the Container Object Paradigm in the Knowledge Repository

---

[9] This capability is sometimes referred to as "dynamic schema evolution."

The knowledge repository also provides facilities for "merging" the representations of two real-world objects once they have been recognized as representations of the same real-world object. For example, merging the real-world objects RWO1 and RWO2 in Figure 14 would lead to the real-world object RWO3 of Figure 13.
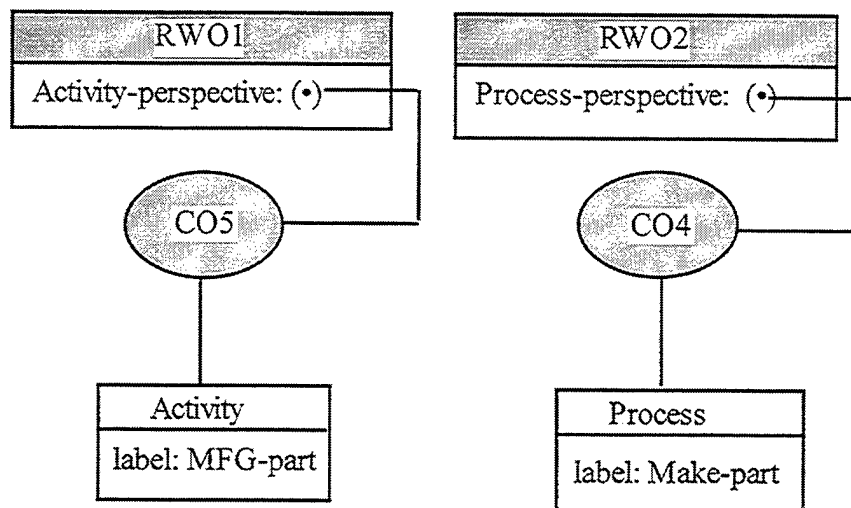
Figure 14

State of the Knowledge Repository Prior to the Merging of RWO1 and RWO2

While the knowledge repository provides a robust data representation scheme, the constraint propagation system (CPS) is responsible for maintaining consistency in the knowledge repository, propagating information, and enforcing constraints. It is a truth maintenance system which has the capability to compute numerical values, deduce truth values, and manage universally quantified expressions.

The CPS has an associated data structure called a "CPS-net." The CPS net contains the information necessary for deducing new information and detecting conflicts. When a new piece of information is entered in the knowledge repository, the CPS is informed of the change in the knowledge base. The CPS then searches the CPS-net for constraints affected by the new piece(s) of information; if any are found, the CPS tries to deduce a new piece of information and/or checks the knowledge repository for potential conflicts resulting from the newly asserted knowledge.

The CPS-net consists of data structures called "cells" and relation objects representing the relations involved in one or more constraints. A cell can represent either a constraint or a statement that is part of a constraint. Constraints are propagated in the CPS when a change occurs in the knowledge repository. Changes in the knowledge base that can trigger a reaction from the CPS include the creation of an object, deletion of an object, modification of an attribute's value, or assertion of a constraint or axiom.

When a constraint is asserted to the CPS, cells are created for that constraint and for its immediate sub-components (ISCs). If the ISCs are complex expressions, cells are created for their ISCs recursively until only

simple expressions remain (i.e., expressions with no ISCs). If one of the simple expressions that composed the constraint denotes an attribute value of an object, a cell is created for that object's attribute. Each cell contains the information necessary to track down the justification for the value of the cell and the consequences associated with modifying the cell's value, together with other information needed by the CPS to propagate changes and detect inconsistencies. When the value of a cell is modified, the cell sends a message to its associated relation (if it has one) informing it of the change. The relation is then responsible for deducing new values and detecting conflicts. If the cell does not have an associated relation (in the case of a simple expression), it sends a message to its immediate ancestor components (if it has any), which are responsible for propagating the change to their associated relation, and so on.

The ability to deduce new information from a constraint is facilitated by the specification of constraint-related information for the relations involved in one or more constraints. This particular information is contained in the "relation object" data structure of the knowledge repository. Relation objects store the knowledge necessary to propagate changes and detect conflicting information. They receive messages from cells in the CPS and send messages to CPS cells to update their values. Relation objects may have associated properties and universally quantified axioms that are used to deduce more information and detect conflicts. When a relation object receives a message from a cell, it checks its associated set of properties and universally quantified axioms to try to propagate additional changes or detect conflicts.

Upon detection of a conflict, the CPS retracts the assertion that is at the origin of the conflict (to ensure the knowledge base is in a stable state) and sends a message to the component that made the assertion. The message provides both an explanation of the conflict (under the form of a list of assertions involved in the conflict) and a suggestion for resolving the conflict. The CPS provides the capability to retract assertions (which causes the corresponding expressions to take the value "False"). Users can resolve conflicts by retracting one or more assertions.

Throughout this process, computational explosion is a major concern for the Level 3 IDSE. The main source of possible computational explosion is the use of quantified constraints. While the CPS implements universally quantified constraints (i.e., constraints of the form "*for all* processes in model M, the process duration cannot exceed 2 hours"), it supports but does not implement existentially quantified constraints (i.e., constraints of the form "for all processes in an IDEF3 model, *there exists* an activity in an IDEF∅ model"). In other words, the system will be able to parse existentially quantified constraints, but the behavior of such a constraint will have to be defined by the user. Existentially quantified constraints are not implemented simply because the expected behavior of the system when enforcing quantified constraints differs from one user to another. Consider the example given above: "for all processes in an IDEF3 model, there exists an activity in an IDEF∅ model." Suppose this constraint has been asserted to the CPS and that a new IDEF3 model is being stored in the ESD. Every time a new IDEF3 process is created, the CPS will fire and find that the quantified constraint has been violated because it cannot find a

corresponding activity. It would then come back to the user, describe the problem, and allow the user to correct the problem (in this case, create a corresponding activity). Clearly, this would be extremely slow and cumbersome for the user.

Another solution would be to create a corresponding activity when a new process is created. However, this solution is not acceptable because of its computational implications. The problem is that this alternative would permit the enforcement of a constraint in the CPS to result in the creation of a new node in the network maintained by the CPS, thus making it impossible to guarantee that the CPS will eventually stop deducing information and reach a stable state. In others words, allowing such actions to result from the enforcement of a constraint increases the chance of a computational explosion in the system.

Existentially quantified constraints such as the one described above typically originate from the need for automatic model generation and translation. However, such services can be more easily and more efficiently provided by dedicated applications. The IDSE architecture supports the automatic and transparent request for services of applications registered with the IDSE. Hence, the behavior described in the example above could be obtained by developing an application to translate an IDEF3 model into an IDEF∅ model, and calling this application after the assertion of each IDEF3 model in the ESD. This implementation would be both more efficient and more flexible because all rules defined explicitly for the translation would be regrouped into one application instead of being distributed over the CPS network.

In general, the CPS will behave well computationally if the number of quantified constraints is kept relatively small. As the number of quantified constraints increases, so does the probability of a computational explosion in the constraint network.

ESD Browser. Each tool integrated with the Level 3 IDSE provides a filtered view of the objects and relations maintained in the ESD. For example, the IDSE-ProCap tool provides a view of the ESD from the IDEF3 perspective, while the IDSE-AI0 Win tool provides a view from the IDEF∅ perspective. The ESD browser provides a global view of the information stored in the knowledge base by enabling direct access to the objects, their types, and the relationships in which they are involved. Through this direct access to the ESD, schema and object attributes can be deleted, added, or renamed instantaneously; schema and objects can be dynamically created or deleted from the database; constraints can be asserted; and values of expressions can be modified.

The ESD browser operates by interacting directly with the ESD. Whereas tools and the ESD exchange data through ISyCL and the ISyCL projector, the ESD browser manipulates the ESD data directly. Since ISyCL is not involved, the ESD browser has access to information across multiple perspectives. For example, the user could view an element originally defined in IDEF∅ while also viewing an element originally defined in IDEF3. With this broader perspective on the knowledge base, the user can create and define more complex relations and constraints.

__Dispatcher.__ The dispatcher is the mechanism through which the various components of the Level 3 IDSE communicate. Operation of the dispatcher is based on a star network connectivity architecture (Figure 15).
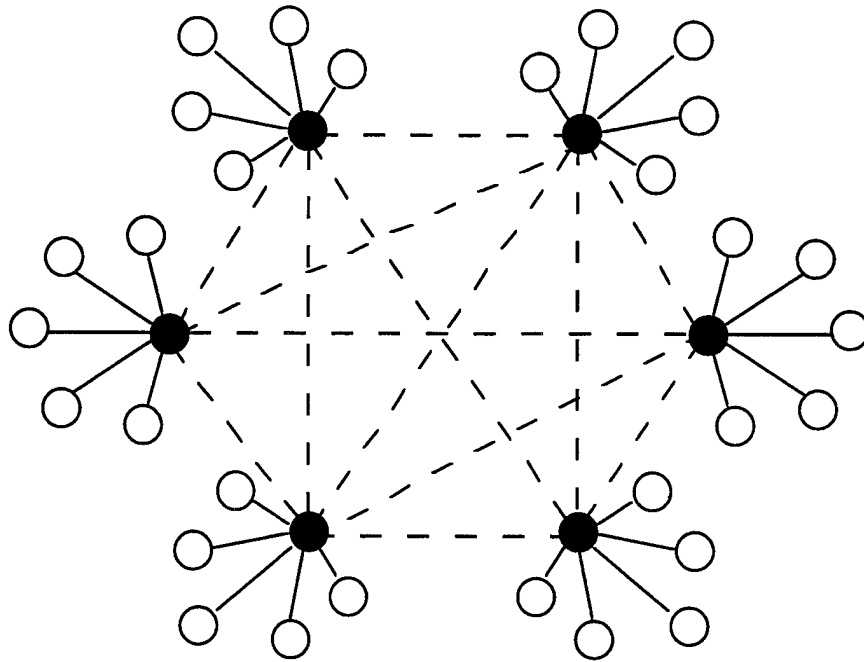
Figure 15

Network Connectivity in the IDSE

Multiple dispatchers, represented by the solid circles in Figure 15, can run simultaneously within this architecture. Each dispatcher is directly connected to a number of IDSE components and to other dispatchers. When a dispatcher receives a request from an IDSE component (e.g., the ESD CSC, a modeling tool), the dispatcher determines the component to which the request is addressed and whether that component is directly connected to it (in which case the component is considered a local host) or is connected to another dispatcher (in which case the component is considered a remote host). If the request is to be handled by a remote host, the dispatcher routes the request to the dispatcher to which the remote host is directly connected. The remote dispatcher then routes the request to the appropriate component.

The dispatcher uses a peer-to-peer communication paradigm rather than a client-server paradigm. Each communication request is assigned a unique key so that the results can be matched with the originating request. The basic unit of communication is a packet containing appropriate routing information, such as the source and destination of a command. There are three packet types: messages, replies, and handshakes.

- Messages: A message is a sequence of commands and submessages. It contains variables that are used to pass data between commands and between peers.

- Replies: A reply can contain informational messages (e.g., error, warning, or status messages), can return message variables, or can be completion notifications.

- Handshakes: A handshake is a connection initiator. It is intended to signal a successful connection between two IDSE components.

Each type of packet is implemented using a host-independent data representation. It is through this data representation that the IDSE is capable of exchanging data and information across different hardware and operating system platforms.

Level 3 IDSE Scenario. Level 3 IDSE implementation has been demonstrated in the context of a software development process. Specifically, a scenario was developed in which a software system was being developed under DoD Standard 2167A. In this scenario, IDEF∅ activity models and IDEF3 process models were to be developed; these models were then to be supported and interrelated with background knowledge provided by 2167A.

In configuring the Level 3 IDSE for this scenario, several ontologies had to be developed. First, the tool ontologies for the IDEF∅ and IDEF3 tools had to be developed so the tool registration process could be completed. Additionally, an ontology of DoD 2167A was developed and asserted to the ESD. With this ontology, relationships to the 2167A ontology elements could be established as models were asserted to the ESD.

The overall functionality of the Level 3 IDSE represents significant advancement in the concepts of integrated environments. However, the role of ontologies, the ESD, and the ISyCL Projector represent the most significant accomplishments of the Level 3 IDSE. Ontologies provide a neutral medium for expressing knowledge about a domain and the value of ontologies is demonstrated by the reliance of the ESD and the ISyCL Projector on ontologies. The ESD not only provides a flexible knowledge representation capability but couples that representation with constraint propagation to ensure consistency in the knowledge maintained within the repository. Finally, the ISyCL Projector provides the ability to dynamically extend the schema of the ESD by parsing an ontology and creating the necessary structures within the ESD to capture information consistent with the original ontology. This capability allows the Level 3 IDSE, and specifically the ESD, to be dynamically extended to accept and manage knowledge captured and provided to the IDSE by new tools.

Experimental Tools Thrust Summary

This chapter has presented the results of the experimentation with IDSE under the Experimental Tools Thrust. Through the various versions of IDSE, the environment attempted to address the issues surrounding project team coordination and integration. The major issues addressed by each IDSE build are summarized below.

- The Level Ø IDSE focused on providing a central repository for system development artifacts, and providing access, version, and configuration control over those artifacts.

- The Level 1 IDSE focused on distributing artifact management across heterogeneous computing platforms, integrating data and applications across these computing platforms (integration services manager), and supporting workgroup and development process control and coordination (framework processor).

- The Level 3 IDSE focused on the knowledge sharing and integration through advanced data representation, data interpretation, and data consistency management.

The Level Ø and Level 1 IDSE concepts are being validated as elements of both the Level Ø IDSE and Level 1 IDSE are currently being addressed by industry. Specifically in the design and engineering domains, applications for design artifact management are being released and marketed. Additionally, applications for workflow management are beginning to have success in the marketplace. Where these applications are presently lacking is in the seamless integration of artifact management and workflow management into a cohesive environment that supports the system development process. This particular shortfall will be minimized in the future as infrastructure technology based on industry standards such as OLE, OpenDoc, DCE, and CORBA mature and enable sharing of data and functionality across applications and platforms.

The Level 3 IDSE demonstrates the greatest potential for future impact. The issues surrounding the Level 3 IDSE will require continued research and maturity because computability and consistency issues will continue to be a problem. However, the concepts of the Level 3 IDSE fit nicely into the still-evolving operating system and "middleware" advancements currently under development (OLE, OpenDoc, DCE, and CORBA). Coupling the data interpretation and consistency management functionality with the representation and distribution capabilities of these emerging technologies would result in powerful knowledge-sharing environments applicable to system analysis, system design, and design rationale capture.

APPLICATIONS THRUST

Introduction

The Applications Thrust also served as a key mechanism for IICE technology demonstration and experimentation to address customer needs. Thus, the purpose for this thrust was to test and refine research results by applying them to real-world problems.

Two major application demonstration efforts were undertaken. The first effort applied the IDEF3 Process Description Capture Method to map Air Logistics Center (ALC) processes, the IDEF4 Object-Oriented Design Method to integrate object-oriented systems design activities, and the IDSE to manage process improvement project artifacts (e.g., AS-IS and TO-BE models). The second effort applied the IDEF3 Process Description Capture, IDEF5 Ontology Description Capture, and IDEF9 Business Constraint Discovery methods to define requirements for a prototype E-3 Programmed Depot Maintenance (PDM) planning support system. The IDEF1 Information Modeling, IDEF1X Data Modeling, and IDEF∅ Function Modeling methods were also applied. IDEF∅ was used together with IDEF3 and IDEF5 to help scope the application demonstration effort. IDEF1 was used with IDEF5 to isolate the information needed for PDM planning and scheduling; IDEF1X was used to design a prototype database to house that information Both application demonstration efforts were conducted at Oklahoma City Air Logistics Center (OC-ALC).

This section first summarizes the most significant accomplishments and contributions of the IICE application demonstration efforts, then describes the two major application demonstration efforts.

Significant Accomplishments

The accomplishments realized through the Application Thrust activity can be described in terms of technological advancements gained through practical application experience, the degree of peer acceptance and user adoption of the IICE products, and the commitments made by OC-ALC to expand IICE technology application efforts. The work at Tinker AFB helped identify additional factors that could improve IICE methods and tools. User acceptance of the IICE technologies ranged from small-scale use to ALC-wide application. Most notably, training in the IDEF3 Process Description Capture method was provided to representatives from every directorate of the ALC and was successfully applied by Air Force personnel to document a wide range of administrative and production processes. Integration experts and standards committee members reused IICE program results in national and international standards products (see discussion in Integrated Systems Theory thrust section of the report). Finally, the impact of the IICE technologies on the day-to-day activities of the OC-ALC workforce generated widespread IICE technology visibility and support to continue additional IICE technology application activity.

The accomplishments of the Applications Thrust activity are summarized below:

1. Demonstrated the relevance, utility, and payback potential of the IDEF3 Process Description Capture method.

2. Conducted an integrated application of the above IICE methods together with previously developed IDEF methods, thereby demonstrating how they complement and support one another.

3. Developed a large base of IDEF3 users at OC-ALC by providing IDEF3 training and method application support to ALC personnel. Nearly 70 individual IDEF3 process description development efforts, representing various segments of the ALC enterprise, were initiated as a direct consequence of these training efforts.

4. Demonstrated an IDEF model-driven approach to design and develop three versions of a prototype PDM planning support system (ProPlan). The prototype was designed to maximize planning data reuse. This design approach was developed to help planners effectively manage the thousands of operations and associated resource definition and precedence constraint data used to plan each workload type.

5. Designed, developed, and demonstrated a self-maintaining, operation-level simulation model supporting multi-aircraft, finite-capacity schedule generation and testing. That is, users can either test the feasibility of a proposed schedule or generate a candidate schedule with the software. Self-maintenance of the simulation model is accomplished by using operation set definitions and user-defined scheduling and simulation goals to configure the simulation model in real time. The IICE team also demonstrated the ability to generate candidate schedules to accommodate required changes, or to support contingency planning and what-if analyses in a design of experiments setting.

First IICE Technology Application Demonstration Effort at OC-ALC

The first IICE technology application demonstration effort at OC-ALC centered on the use of four IICE program products: IDEF3 Process Description Capture method, IDEF4 Object-Oriented Design method, and the Level Ø IDSE. The specific tasks undertaken as part of the application demonstration effort are summarized in the following list:

1. Identify the high payoff potential of the application of IICE technologies to address OC-ALC needs.

2.   Provide IDEF3 Process Description Capture and IDEF4 Object-Oriented Design Method training and application support to OC-ALC personnel.

3.   Install and demonstrate the Level Ø IDSE.

This approach made extensive use of IDEF3 method training to generate OC-ALC personnel participation in the project. This strategy leveraged the ease of use and process orientation of IDEF3 to speed technology transition and demonstration efforts. OC-ALC personnel were trained in the use of IDEF3 (Task 2) and provided direct assistance in applying IDEF3 to document processes. Two OC-ALC personnel were also trained as IDEF3 trainers and assisted with IDEF3 training courses. This activity helped accelerate IICE team efforts to identify high-payoff opportunity areas for IICE technology use (Task 1), demonstrate the effectiveness of IDEF3 (Task 2), and refine and validate OC-ALC's model integration needs (Task 4). Training and the subsequent assistance provided to OC-ALC personnel resulted in the development of nearly 70 individual IDEF3 process descriptions representative of the diverse set of processes which compose the ALC. These process description capture efforts were used by OC-ALC personnel to identify process links between organizations, identify non-value-added activities, facilitate manpower planning studies, capture process knowledge prior to personnel losses, prepare training materials describing ALC procedures, and so forth.

The utility of IDEF3 and IDEF4 was also demonstrated by using these methods to tailor the Level Ø IDSE for application demonstration at OC-ALC. The Level Ø IDSE was used as a model repository to store, track, configuration, manage, and integrate a range of process mapping activities being accomplished by OC-ALC personnel. Tailoring the IDSE for this purpose involved rehosting the experimental version, developed on the Symbolics machine, to the Macintosh platform. IDEF3 and IDEF4 were used to tailor and re-host the experimental version of the Level Ø IDSE. IDEF3 was used primarily to redesign IDSE Level Ø user interaction scenarios. IDEF4 was used primarily to design object classes used in the new implementation. Level Ø IDSE rehosting was completed in four weeks. Use of IDEF3 and IDEF4 enabled both rapid re-engineering and re-implementation of the IDSE concepts into a specifically targeted application.

Task 4 identified some of OC-ALC's method support and model integration needs. The process, information system, and organization perspectives are each the subject of analysis and design through independent, but mutually supportive, modeling methods. Methods, and the models that result from their application, provide a medium through which change can be explored, implemented, and monitored. Such environments are said to be *model-driven* and require that changes to models reflecting one perspective of the business be accurately and efficiently propagated across other models of the business. These environments also require that consistency be maintained across models. These needs guided the IICE team's study of OC-ALC's modeling and model integration needs. IICE project developments and findings in model data representation, model data interchange, and model data interpretation were analyzed for their relevance to OC-ALC needs. Both existing and proposed

IDEF methods were also analyzed to determine the extent to which they address the needs of OC-ALC's complex business, engineering, maintenance, and manufacturing systems.

Accomplishments of the Effort

The accomplishments realized through this application demonstration effort are listed below.

1.   Demonstrated the utility and payback potential of the following IICE products and technologies:

    a.   IDEF3 Process Description Capture Method.

    b.   IDEF4 Object-Oriented Design Method.

    c.   IDSE architecture and Level Ø implementation.

    d.   NIRS, which provides a formal foundation for modeling language construction, model data interchange, and model data interpretation.

2.   Generated a new base of IICE product users at OC-ALC.

3.   Accelerated documentation of Tinker AFB's processes by:

    a.   Providing IDEF3 training to OC-ALC personnel which helped achieve ALC-wide buy-in for IDEF3. Also helped establish IDEF3 as OC-ALC's primary method for process and improvement activities..

    b.   Establishing an organic IDEF3 training capability at OC-ALC.

    c.   Providing consulting support and assisting in the development of nearly 70 individual IDEF3 process descriptions spanning workload planning; material acquisition; inventory management; two-level avionics maintenance; shop scheduling; equipment management; purchase order processing; engineering data management; Technical Order (TO) stock, store, issue, and update; PDM; and manufacturing process planning.

4.   Provided object-oriented systems design and management training to 15 Tinker AFB Management Information System (MIS) personnel. Training included exposure to object-oriented methods, programming languages, and database technologies. Course participants were introduced to the IICE IDEF4 Object-Oriented Design Method and instructed on its use as a management tool for OC-ALC system development projects.

5. Provided OC-ALC with a model repository and configuration management environment (Level Ø IDSE) specifically tailored for OC-ALC process improvement project needs.

6. Identified five IICE technology application demonstration areas supporting OC-ALC needs. Application demonstration opportunity areas identified include engineering data management; TO data management; reverse engineering; process planning; and KC-135 corrosion data management. For each opportunity area, known operational needs and problems were described, application demonstration tasks were recommended, and applicable IICE products and technologies offering unique solutions were identified.

Second IICE Technology Application Demonstration Effort at OC-ALC

The second IICE technology application demonstration effort at OC-ALC addressed an opportunity area identified by OC-ALC/LAP. The subject area chosen was aircraft depot maintenance. More specifically, the effort was scoped to address planning, scheduling, and control needs for ALC PDM operations. IICE product application was directed toward constructing a prototype planning support system (ProPlan) through which to demonstrate proposed process and information system change opportunities.

A number of factors influenced the Air Force's decision to pursue this application demonstration effort. First, aircraft PDM is the primary workload of the Air Logistics Center. It is also the largest workload. Improvements to the processes supporting this workload therefore offered more payback potential. Second, ALC management was challenged with making improvements to the PDM line and reorienting current practices to support both contract (commercial sector) and organic (public sector) operations using the same people, facilities, equipment, and information. A key motivation was the fact that PDM workload, previously the sole purview of the ALC, was being challenged in open commercial competition. OC-ALC was awarded a contract in full-and-open competition to service the E-3 fleet. Existing systems also provided limited, poor, or no support in areas required to manage and control PDM activities effectively.

These systems are part of an extensive network of data systems that govern OC-ALC's depot maintenance operations. In all, more than 30 individual systems compose the depot maintenance data systems network. These systems range in functional responsibility from material requirements projection to inventory tracking, and from maintenance operations planning to expense reporting. Each has evolved semi-independently to serve the needs of their respective communities. The cumulative result is a patchwork of old and new systems that perform a wide range of data collection, monitoring, and control activities.

Today's depot maintenance data systems struggle to meet the needs of an environment requiring new levels of ALC infrastructure agility and economy. Increasingly, the explosive rate of change to which these systems must

adapt creates an environment where ALC organizations can become reactive, inflexible, slow to respond, and costly. Symptoms that are typical of organizations falling prey to these challenges are as follows:

- **Process owners do not own their process data.** Process owners frequently must go to other organizations to gain access to critical data. Having no ownership of the data, users often create parallel systems to maintain the data they need. Since this data is locally maintained it is inaccessible to others who may find it useful.

- **Accessible data is often unusable.** Data collection systems often maintain only summary-level information, rendering it largely unusable to all but high-level management.

- **Users often feel compelled to circumvent data systems.** As the pace and scope of process change accelerates and software maintenance and development backlogs users often attempt to use the data systems for purposes those systems are not designed to support, or they begin to ignore the command data system completely. In either situation, data available to downstream users becomes increasingly suspect and/or unavailable.

- **Automated data systems drive the process and stifle positive change.** Policies established to ensure that downstream data users get the data they require breeds an environment in which data systems become the masters and users their slaves. When this inversion occurs, the burden of meeting demanding data system needs may come at the expense of activities that promote creativity and innovation.

The purpose of this effort was to demonstrate the potential of the IICE technologies to facilitate simultaneous organizational, cultural, process, informational, and technological change. The selected area of demonstration was the systems supporting E-3 PDM. Among the specific needs identified were the following:

1. Reduce the overall cycle time and cost of the PDM process.

2. Improve the responsiveness of PDM support processes.

3. Increase the accuracy of projected resource and material needs.

4. Improve the realization of promise dates.

5. Decrease the number of discrepancies during final post-dock test.

6. Reduce the amount of unplanned, unscheduled, and over-and-above (O&A) work in the PDM process.

The distinction between production maintenance and production manufacturing systems became very important as the IICE team worked to provide viable solution concepts using the IICE technologies. Systems that support depot maintenance operations are based on the assumption that production maintenance is roughly the same as production manufacturing. This is particularly true of depot maintenance planning and scheduling systems. The difference between production maintenance and production manufacturing can be characterized in terms of the amount and type of work to be done. Planners in the production manufacturing environment can easily determine the work needed produce a given product, although the number of products to produce is often unknown. Planners in a production maintenance environment, like those in E-3 planning, know the requirements for annual aircraft throughput. However, they don't know what additional work requirements will surface while mechanics perform planned operations. Both environments place different demands on the systems used to support production — including the operation planning, material acquisition, and scheduling systems.

As is typical of production manufacturing planning, OC-ALC's material planning systems presume that the BOM (required for maintenance) can be determined beforehand (i.e., is predictable). However, historical patterns demonstrate that between 20 and 35 percent of depot maintenance work is unpredictable or "O&A." Second, material requirements computation assumes one can accurately predict future requirements by analyzing past usage. For predictable items (i.e., spare parts whose replacement rates are relatively constant), the latter assumption is perfectly good. However, it is interesting to note that approximately only 20 percent of the items managed by these systems are considered predictable items. Fortunately, those items supply the material for most depot maintenance work (65 to 80 percent of depot workload). The remaining 80 percent of these items fall into the unpredictable group (Figure 16). Both predictable and unpredictable items are negotiated, scheduled, and production-tracked by line item. However, unpredictable items are managed at an aggregate level when it comes to the allocation of manpower, job order control, and financial accounting.

Consequently, as much as a third of all aircraft maintenance material needs are unpredictable. Further complicating matters is the fact that as the aircraft ages, the frequency of unpredictable and O&A work (e.g., through the discovery of cracks and corrosion) increases. This phenomenon serves to both raise the ALC's depot maintenance workload and increase the ratio of unpredictable to predictable work.

The high rate of unpredictable and O&A work places additional burdens on the material acquisition and scheduling systems requiring increased flexibility and responsiveness. This phenomenon also places increased importance on the provision for timely, accurate workload and resource status information. Policy and procedural constraints, a shrinking base of suppliers for aging aircraft parts, and growing needs for replacement items combine to place enormous burdens on an already stressed material acquisition system. Scheduling systems capable of rapidly adapting to changes in resource availability and work requirements are not available. Today's command data systems providing scheduling support systems perform well for master scheduling (i.e., scheduling base-level input and output schedules for aircraft). However, these same systems provide little support for more fine-grained

scheduling. The fact that planners and planning management are the only users of these systems further demonstrates the validity of the conclusions. Finally, workload status is maintained through relatively undisciplined operation status reporting and manual data-entry processes. Resource status information, if available, is distributed across systems that are not integrated with the shop floor feedback and scheduling systems.
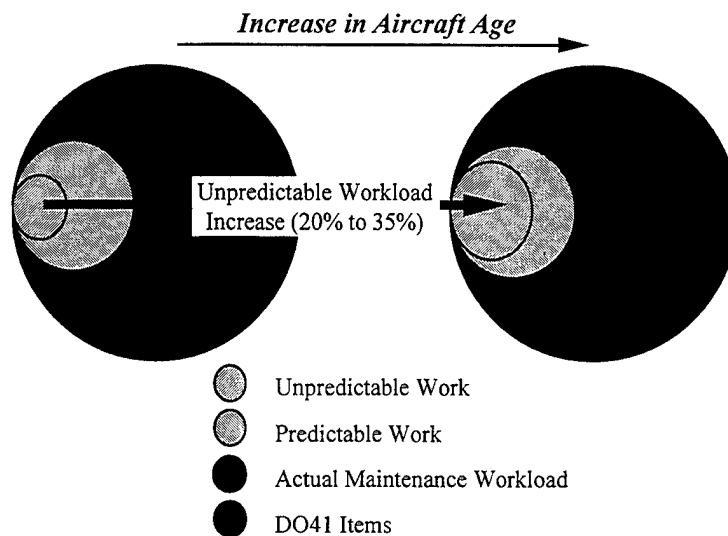
Figure 16

Unpredictable Workload Creates Material Requirements Computation Uncertainty

It soon became apparent that a production maintenance-oriented, closed-loop system for PDM requirements definition, planning, scheduling, and control would need to be developed. To attempt the complete development would have been beyond the means and time available. However, having this long-range goal in mind helped lead the IICE team to develop and demonstrate key portions of the total solution. The specific tasks are listed below.

1. Define, design, construct, and demonstrate a prototype planning support system (ProPlan). Demonstrate the use of IICE technologies in developing ProPlan.

2. Provide IICE method training to targeted users of the ProPlan system to involve them in its development. Also, provide training in the use and maintenance of prototype software developments (i.e., ProPlan, prototype database, major-job-level simulation) and models (i.e., information and process models).

3. Develop operation-level process descriptions for a representative subset of E-3 PDM operations.

4. Develop a high-level (i.e., major-job level) simulation model of the E-3 PDM process.

5.   Identify shop floor data collection requirements.

6.   Develop a prototype database, designed to meet the requirements identified in Task 6 above, to demonstrate the storage and use of shop floor information.

7.   Develop recommendations on the hardware, software, and strategies for collecting shop floor data using bar coding technology.

The resulting products of these efforts are listed below:

1.   Three software versions of the prototype planning support system, ProPlan.

2.   ProPlan software users manual.

3.   ProPlan programmers manual.

4.   IDEF3 process descriptions, IDEF5 ontology descriptions, IDEF1 information models, IDEFØ function models, and IDEF1X data models reflecting the AS-IS analysis and TO-BE design work accomplished.

5.   Operation-level process description (IDEF3) of Analytical Condition Inspection (ACI) Task 1067 operations.

6.   E-3 PDM high-level simulation model and report.

7.   Prototype Stochastic Resource Requirements Projector (SRRP) enabling both multi-aircraft schedule testing and schedule generation.

8.   Shop floor information requirements model (IDEF1) and database design (IDEF1X).

Approach

The long-term vision of a closed-loop planning, scheduling, and execution environment is depicted by the IDEFØ diagram in Figure 17. The main topic of application demonstration was the *Plan* function. Additional effort was also planned to explore the interfaces between the *Plan* function and the *Schedule* and *Execute* functions. However, application demonstration in these two areas was more limited than that planned for the *Plan* function.

71
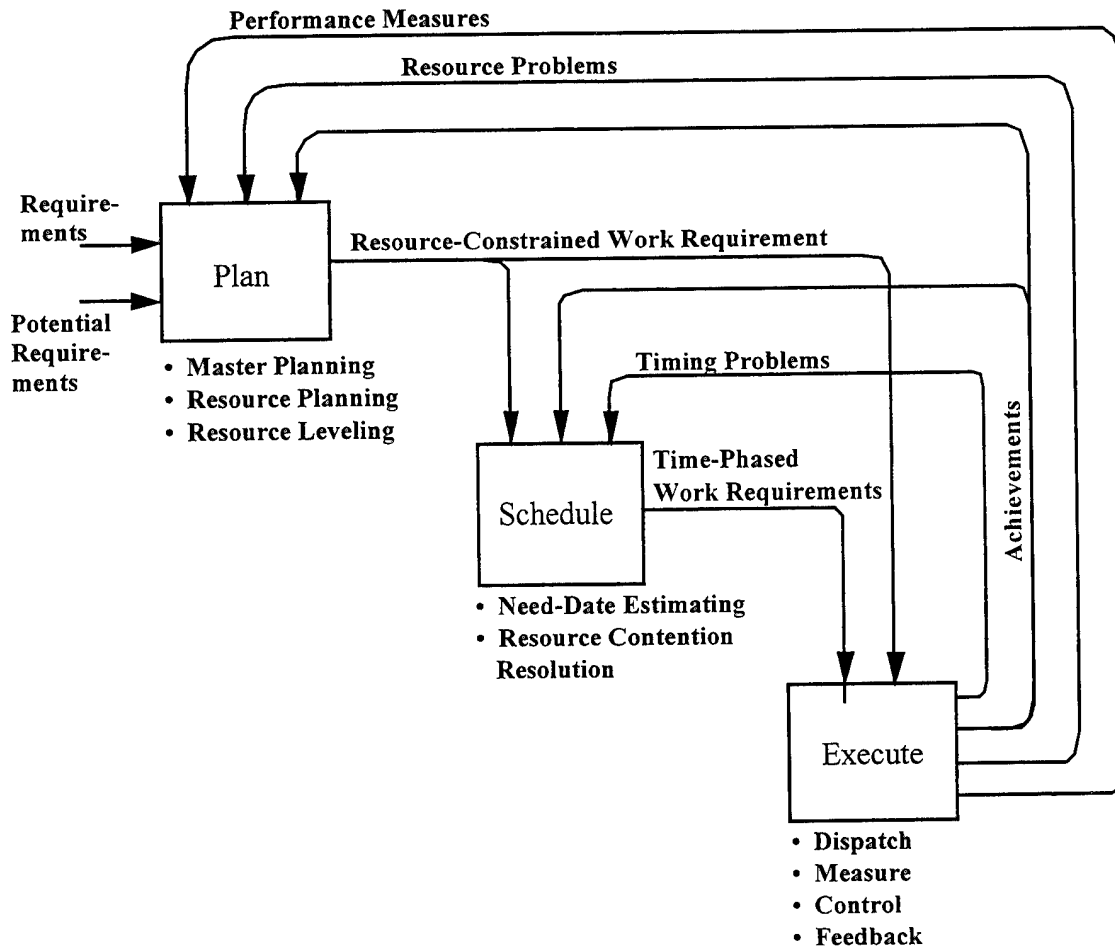
Figure 17

Function Model of a Closed-Loop Production Maintenance System

With this scope in mind, a model-driven approach to requirements definition and prototype system development was outlined. Integral to this approach was extensive use of the IDEF3, IDEF5, and IDEF9 methods. These methods were applied together with previously developed IDEF methods (i.e., IDEFØ, IDEF1, IDEF1X) to support additional analysis and design decision-making activities, and to demonstrate how the IICE methods complement previously developed IDEF methods. Of the methods applied, IDEF3, IDEF5, and IDEF1 were used most extensively. These methods were used primarily for analysis, requirements definition, and TO-BE process design. IDEFØ was used with IDEF3 to help scope analysis efforts. IDEF1X was used to translate IDEF1 information management requirements into a logical database design of the prototype shop floor database.

The first task was to identify the key scenarios associated with PDM planning, scheduling, and execution. Through this activity, the events, activities, and processes that characterize both normal and exceptional situations in PDM planning, scheduling, and execution were identified and categorized. The resulting list of scenarios was then used to refine the initial project scope further. For example, only a select set of scenarios under the *execute* function

72

were chosen for subsequent modeling and analysis. This list of scenarios was also used to establish data collection, analysis, and description and model development priorities.

Once the key scenarios were identified, we began investigating how those scenarios were currently performed, the reasons for current success, and the areas in need of improvement. As with any attempt at process, information system, technological, and cultural change, this kind of analysis requires a thorough understanding of the current, or AS-IS, environment. The AS-IS environment is a mixture of both the formal (or "as-designed") process and the informal (or "as-practiced") process. Gaps arising over time between the formal and informal system offered clues about existing system inadequacies and opportunities for improvement. Both formal and informal system descriptions were developed — first, using IDEF3 to document PDM planning, scheduling, and execution processes; second, using IDEF5 to document concepts, terminology, and relationships unique to those processes. IDEF1 was also used to study the information managed to support the AS-IS process. IDEF1 modeling was accelerated by reusing the results of ontology analysis (IDEF5), when available. The techniques embodied in the prototype IDEF9 Business Constraint Discovery method were also used to identify outdated assumptions and policies, identify enabling and limiting constraints with respect to long-term PDM process improvement goals, and explore cause-effect relationships. Analysis of the AS-IS environment also provided a framework to measure potential gains through adoption of proposed reengineering changes at a later time.

AS-IS analysis also helped to surface process and information system improvement opportunities. Simply involving ALC personnel in documenting their processes helped them think about and articulate how things might be done differently, or better. This phenomenon is largely due to the fact that humans tend to think inductively; that is, we use a pre-established mental image of how a process is or should be done as the springboard for developing a model of the process. One's mental model is altered, and subsequently documented, through the discovery of previously unknown facts. The IDEF3 method provided a uniquely well-suited mechanism for discovering this process-related knowledge. Directly involving ALC personnel in this process helped to leverage their intuitions about what could be improved and how.

These intuitions were used to explore both radical and incremental process improvement opportunities. Incremental improvement techniques attempt to institute positive change while maintaining the basic design of the original process. This kind of improvement strategy relies heavily on domain expert intuition and experience to streamline processes, eliminate redundant activities, and/or speed up processes through automation. Radical or large-scale improvement techniques seek to change the *basic model* of doing business. In short, these techniques search for opportunities to initiate a *paradigm shift* through the elimination of entire processes and the artifacts or products of those processes. The search for paradigm shift opportunities required looking outside the current realm of ALC system experience to identify comparable systems or processes that perform the same function in entirely different and innovative ways.

The process improvement opportunities identified through these questions were then used to formulate candidate TO-BE processes using the basic guidelines depicted in Table 2. The IDEF3 method was also used during this process to help OC-ALC personnel with this conceptualization and envisioning process.

Table 2. Guidelines Used for TO-BE Process Formulation

Guidelines for TO-BE Process Formulation
Organize processes around outcomes, not tasks.
Have those who use the output of the process perform the process.
Identify redundant creation, storage, and/or use of information.
Subsume information processing work into the real work that produces the information.
Capture information once, at the source.
Treat geographically dispersed resources as through they were centralized.
Link parallel activities instead of integrating their results.
Put the decision point where the work is performed and build control into the process.
Identify portions of the AS-IS process that exist to enforce constraints that are no longer applicable.
Eliminate non-value-added activities.
Simplify, consolidate, streamline, and make parallel value-added activities.

Collectively, these efforts produced a set of IDEF3 process descriptions representing ALC PDM support processes as designed, as practiced, and as envisioned for the future (Figure 18). Companion IDEF5, IDEF1, and IDEF1X models were also developed as needed.

TO-BE processes were then used to develop requirements for the prototype ProPlan system, shop floor database, and scheduling support mechanisms produced as software demonstrations for the effort. TO-BE processes define the process architecture for prototype software developments. Additional software architectures were then defined; these included function, information, module, network, and menu architectures. The function architecture was developed, in part, through the use of IDEFØ; however, textual definition proved to be more time efficient for this task. Information architecture development was accomplished using the IDEF1 and IDEF1X methods. These models were needed to reflect changes arising through changes to the process. Non-value-added information maintained by the AS-IS information system was identified and removed in the TO-BE model. At the same time, the TO-BE model included additional information not present in the AS-IS but necessary to achieve long-range

goals. For example, existing planning systems in use at OC-ALC maintain no information in the operation set about what facilities or special equipment is required to perform the work specified; yet, this information is critical to effective scheduling. This and other value-added information that were absent from the existing system were included explicitly in the prototypes developed. Module architecture definition involved allocating functions to system components. The role of this activity was to maximize reuse of COTS software to minimize development costs and reduce risk. Several COTS components were incorporated in the demonstration prototype including Microsoft Word (document processing), Microsoft Project (master schedule definition), Microsoft Excel (operation sorting, reporting, and spreadsheet preparation), Microsoft Access and Oracle (database), Witness (simulation), and ProSim (process modeling and Witness code generation). The network architecture was developed informally at both a logical and physical level, and was provided to OC-ALC for use in infrastructure planning and development. Several iterations of menu architecture definition were accomplished using a rapid prototyping approach with the help of component-based software development environments (Digitalk Parts and Visual Basic). The resulting software was then demonstrated and tested incrementally by developing three versions of the software. Feedback from demonstration and testing was used to refine and improve the prototype software.
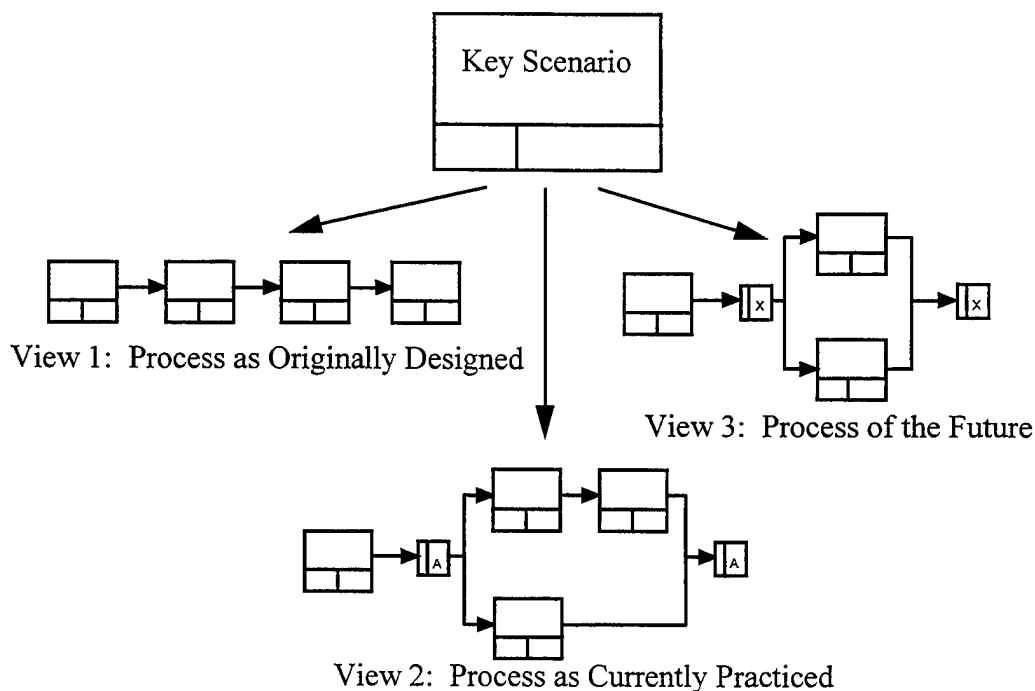
Figure 18

Multiple Views of a Process

75

Accomplishments of the Effort

The accomplishments of this effort are divided into two categories. The first lists goals accomplished and significant innovations; the second presents long-term goals, a representative set of candidate metrics for assessing relative progress, and some measurable benefits in terms of those metrics that may be realized through adoption and use of the demonstrated system concepts. Some of the achieved goals and significant innovations include the following.

1. Successfully demonstrated the relevance, utility, and payback potential of the IDEF3 Process Description Capture and IDEF5 Ontology Description Capture methods.

2. Designed, developed, and demonstrated three versions of the prototype ProPlan system. The delivered system was a multi-user, client/server, network-application-making extensive use of COTS software.

3. Developed a planning support system design based on the concept of maximized planning data reuse and planning constraint management. This design approach helps planners more effectively manage the thousands of operations and associated resource definition and precedence constraint data used to plan each workload type. This design strategy reverses the ratio of time spent on bookkeeping versus analysis and decision-making tasks allowing planners to produce better plans in less time

4. Produced a prototype E-3 shop floor database together with a set of screens and reports characteristic of the E-3's shop floor information system needs. In developing the prototype database, the IICE team demonstrated a model-driven approach to design, develop, and maintain an E-3 shop floor database.

5. Developed simulation-based scheduling concepts for rapid, detailed, and cost-effective operation-level schedule definition in the maintenance environment.

6. Demonstrated the utility of IDEF3 in capturing operation-level precedence information. Also demonstrated the leverage provided by capturing precedence constraints with which to generate multiple possible sequence paths traversing the operation set over the currently used process of defining one possible sequence path. This approach reduces the planning workload by allowing them to create partially linked networks rather than fully linked networks, as is currently required.

7. Designed, developed, and demonstrated the SRRP, which provides a self-maintaining, operation-level simulation model supporting multi-aircraft, finite capacity schedule

generation and testing. That is, users can either test the feasibility of a proposed schedule or generate a candidate schedule using the SRRP. Self-maintenance of the simulation model is accomplished by using operation set definitions and user-defined scheduling and simulation goals to configure the simulation model — in real time. Demonstrated the ability to generate candidate schedules rapidly to accommodate required changes, or to support contingency planning and what-if analyses in an environment tailored for running multiple experiments. This was one of the most significant innovations of the project.

Additional accomplishments, presented in Table 3, reflect measurable benefits that were demonstrated through hypothetical use of the prototype software products of this effort.

Table 3. Goals, Metrics, and Benefits

Goal	Metric	Measurable Benefits
Reduce PDM cycle time	Cycle time losses through duplication of effort and lost economies	Dispatch-based schedule simulation mechanism automatically translates planner-defined operation precedence constraints into candidate schedules that eliminate unplanned duplicate operations and maximize productive use of resources
	Frequency of unproductive resource utilization caused by resource unavailability	When provided with visibility on current resource status, demonstrated the ability to rapidly regenerate both operation and resource schedules to maximize productive resource utilization
Improve responsiveness of PDM support processes	Time to perform planning activities	Two-day manual effort to select tail-specific operations reduced to minutes; opportunity for error reduced by an order of magnitude.
		Overnight print-check cycle required for planning data maintenance and quality assurance eliminated; improved user interface provides an additional 50 percent cycle- time reduction for routine planning tasks.

Table 3. (Continued)

Goal	Metric	Measurable Benefits
	Time to perform scheduling activities	Simulation-model-generated schedules produce a 100:1 decrease in operation scheduling time
	Time to perform what-if analyses of workload and/or schedule changes	Cross-aircraft schedule simulation set-up time reduced from 200 man-hours to minutes; ripple effect of scheduling changes on other aircraft produced in 1 hour versus 24
	Accuracy of schedule projections	Finite capacity schedule generation capability capable of accounting for all resources (the current system is only capable of considering the skill resource); extended current system's ability to perform deterministic schedule projection with a stochastic schedule simulation capability
Improve the accuracy of projected resource and material needs	Percent of the total set of resource requirements that can be specified by planners	Able to specify 100 percent of the resource types required to perform an operation (adding facilities, equipment, and TOs to skill and material call outs); current ability to identify only primary resource options within a resource type (e.g., skill) is extended to permit specifying multiple resource options
	Accuracy of projected resource need dates	Automatic resource leveling supported by finite-capacity scheduling mechanism provides improved resource schedule projections to assist with material acquisition planning, equipment repair scheduling, and so forth
Improve the realization of promise dates	Time required to incorporate additional work requirements (e.g., shakedown and O&A) into tail-specific plans and schedules	Provision for mechanics to input directly shakedown and O&A work requirements into ProPlan has the potential to reduce planning cycle times measured in days and weeks to hours; one-hour versus one-day cycle time to assess schedule impacts

Table 3. (Concluded)

Goal	Metric	Measurable Benefits
Reduce the number of discrepancies during the final post-dock test	Provisions for continuous improvement of operation definitions	Annual maintenance of operation set data is now supported by rapid search, retrieval, and review of operations; extensible, context-sensitive help facilities provide on-line mechanisms for operation design/redesign practice; list editing done "in the blind," line-by-line has been replaced with full-page Microsoft Word text editing facilities
Reduce the amount of unplanned, unscheduled, and O&A tasks in the PDM process	Number and frequency of scheduler requests of planners to incorporate major jobs in tail-specific plans that were inadvertently overlooked	Total elimination of such requests to planners through automatic selection of tail-specific operation sets based on the work specification tasks called for in the work order (i.e., transcription errors eliminated)
	Percent O&A operations maintained in planning system for future planning activities	One hundred percent maintenance of O&A operations (current system allows planners to define but not maintain O&A operation data)

Summary

At the outset of the first technology transition effort at OC-ALC, it would have been difficult to predict the impact of the IICE technology insertion. Currently, IDEF3 continues to contribute to OC-ALC's process improvement efforts. Other IICE technologies have proven their long-term worth. For example, IDEF4's usefulness in rapidly developing applications and helping managers direct object-oriented systems development projects was demonstrated. Demonstration of the Level Ø IDSE illustrated its value as a model repository, configuration management tool, and change management mechanism for environments taking the first steps toward integration.

The second application demonstration effort conducted at OC-ALC was large in both scope and ambition. Its success depended on a thorough understanding of the unique characteristics distinguishing production maintenance from production manufacturing and on a well-integrated team. The IICE methods provided an efficient means to do both. Used in concert with previously developed IDEF methods, the project team worked with ALC personnel to demonstrate successfully a model-driven approach to systems analysis, design, and development. IDEF3 again proved its versatility and power as a process description capture mechanism. IDEF5 and IDEF1 were also used extensively, providing additional and important perspectives on the problem. Although still in a relatively early formative stage, the IDEF9 method also helped the team identify constraints. Furthermore, the effort

generated a self-maintaining, operation-level simulation model supporting multi-aircraft, finite-capacity schedule generation and testing. That is, by using the prototype to perform normal planning activities the tool is able to generate updates to the simulation model of the plan to be executed using the planning data itself. Finally, the effort produced prototype software demonstrating process and information system reengineering concepts offering significant payback potential.

A more detailed description of this application demonstration effort can be found in the report entitled, *IICE Technology Transition Effort for E-3 Programmed Depot Maintenance (PDM) Final Report* (Painter et al., 1995, pending).

ONTOLOGY THRUST

Introduction

In the context of information management, *ontology* is the task of acquiring the terminology, grammar, and sanctioned inferences (i.e., the *logic*) of a given domain and storing them in a usable representational medium. Ontologies provide the background context in which information is transferred between agents. Without an ontology, only data transfer is possible.

Capturing ontological information is especially crucial in the context of information integration of large, distributed systems. Effective corporate information management (CIM) involves coordinating the resources of many different clusters of cooperative organizations. Each cluster makes its own contributions, and the overall success of the project depends on the degree of integration (information and knowledge sharing) among those different clusters throughout the development process. A key to effective integration is a system ontology that can be accessed and modified across clusters, and which captures common features of the overall system relevant to the goals of the disparate clusters. This common framework promotes sharing information that arises from various sources, eases problems of information base maintenance, and enhances the reusability of information.

The relationship of the Ontology Thrust to other IICE focus areas is summarized as follows (see Figure 19):

1. *IDSE*: Ontologies are at the heart of the ontology-driven information integration architecture of the IDSE (Level 3).

2. *IDEF5*: The IDEF5 method facilitates rapid ontology acquisition, development, and maintenance.

3. *Other Methods*: Method ontologies help identify the constraints among different method concepts.

4. *Applications Thrust*: Domain ontologies help in the development and maintenance of integrated information systems. At Tinker AFB, ontologies have played an important role in the application of IICE technology. That is, they have been instrumental in determining the relationships between different functional areas, aiding reverse engineering automated systems, and developing new operational systems.

5. *Integrated Systems Theory Thrust*: Method and domain ontologies provide the basis for the approach to enterprise integration.
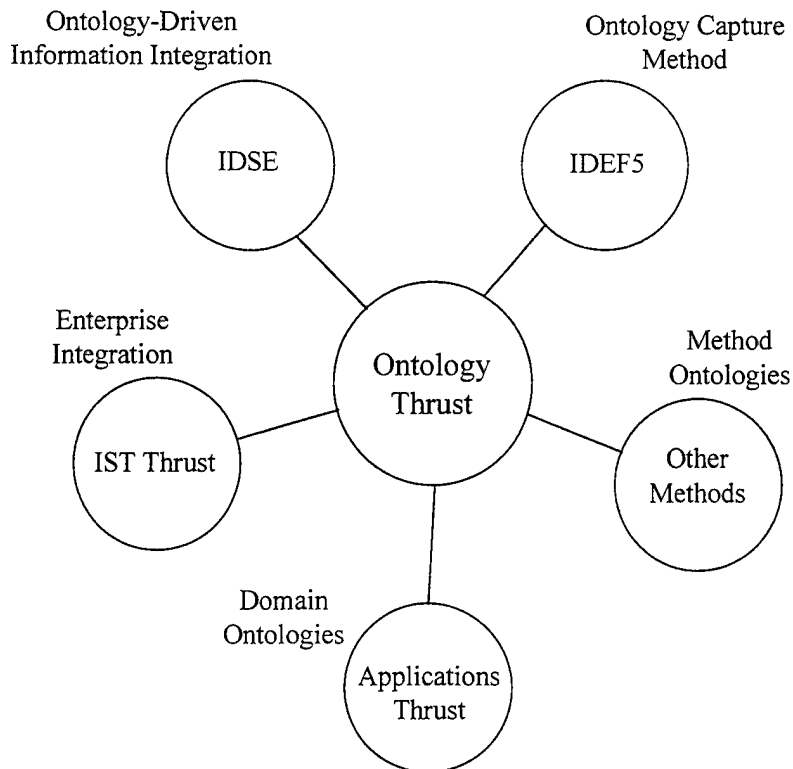
Figure 19

Ontology Thrust in Relation to Other Focus Areas

The overall goal of the Ontology Thrust is to develop theoretical and methodological foundations for capturing and managing ontology information. The main objective of this thrust is to develop realistic bounds for ontology development so well-developed ontologies can be constructed with the available time and resources.

Significant Accomplishments

The main accomplishments of the Ontology Thrust can be divided into three main groups:

1. *The development of domain ontologies*

   Domain ontologies include manufacturing process planning, manufacturing process controller, information modeling methods, BOM, shop floor control systems, space mission planning, and simulation model design.

2. *Theoretical developments*

   Theoretical developments are described in detail in the IDEF5 Method Report (Mayer et al., 1994). The more prominent theoretical developments include the following:

   - Detailed development of the conceptual foundations of ontology.

- Development of a characterization of higher-order properties and relations for the purposes of ontology.

- Incorporation of process kinds and object states into IDEF5. Because of the central importance of process information in characterizing a given domain's ontology, the ability to represent the structure of processes and dynamic object states (in a manner compatible with the IDEF3 method) was incorporated directly into the IDEF5 method.

- Development of the IDEF5 Relation Library. A set of important, commonly identified relations were identified, characterized, and incorporated into the IDEF5 Relation Library. (The Relation Library is incorporated into the IDEF5 Method Report as an appendix.)

3. *Exploration and clarification of the role of ontology in enterprise integration.*

- A detailed description of the role of ontologies in enterprise integration is given in "The Role of Ontology in Enterprise Integration" (Mayer, Menzel, Painter, & Benjamin, 1993).

What is an Ontology?

Any domain with a determinate subject matter has its own terminology — a distinctive vocabulary used to discuss the characteristic objects and processes that constitute the domain. A library, for example, possesses its own vocabulary, one having to do with books, reference items, bibliographies, journals, and so forth. Similarly, semiconductor manufacturing has its own language of chips, wafers, etchants, designs, and the like. The nature of a given domain is thus revealed in the language used to discuss it. However, the nature of a domain clearly is not revealed in its corresponding vocabulary *alone*; in addition, one must provide rigorous definitions of the grammar governing the way terms in the vocabulary can be combined to form statements, and clarify the logical connections between such statements. This additional information is necessary to understand both the natures of the individuals that exist in the domain and the critical relations they bear to one another. An *ontology* is a structured representation of this *ontological* information. More exactly, an ontology is a domain vocabulary complete with a set of precise definitions, or *axioms*, that constrain the meanings of the terms in that vocabulary sufficiently to enable consistent interpretation of statements that use that vocabulary.

Taken by itself, an ontology may not seem much different from a data dictionary. However, a data dictionary is typically a compendium of terms with definitions for the individual terms stated in natural language. By contrast, the grammar and axioms of an ontology are stated in a formal language, with a precise syntax and clear formal semantics. Consequently, ontologies are, in general, far more rigorous in their content than a typical data

dictionary (and, hence, more so than a typical data "encyclopedia" because an encyclopedia is just a collection of related data dictionaries). Ontologies also tend to be more complete as well; *relations* between concepts and objects in a domain, and constraints on and between domain objects, are made explicit, thus minimizing the chance of misunderstanding logical connections in the domain.

A data dictionary, by contrast, generally relies upon an intuitive understanding of the terms in question and the logical connections between the concepts and objects they represent. This works well in small restricted domains, but problems arise when information systems span organizational, geographic, and enterprise boundaries. The traditional approach is problematic for several reasons, one of which is that persons in different domains might understand the same term in subtly different, but important, ways that are not uncovered in a natural language definition (which can lead to inconsistent interpretations of the same term across different contexts) and so forth. The construction of "background" ontologies that capture such differences provides the necessary semantic backdrop for interpreting text (and data, generally) across domains that are connected by large information systems. Furthermore, the discipline of expressing the ontology information in a formal language enhances the skills necessary for extracting the information — in particular, the ability to abstract (1) from particular objects to the kinds of which they are instances, (2) from particular connections to the relations such instances stand in generally, and (3) from particular behaviors to the constraints that bind instances of various kinds together logically within the domain.

Historical and Conceptual Foundations

Historical Background

In traditional Western thought, an ontology is regarded as an attempt to divide the world at its joints, to provide an exhaustive inventory of *what exists*, of what there is. Historically, the practice of constructing ontologies grew from the branch of philosophy known as metaphysics, which deals with the nature of reality. Metaphysics is often associated with questions beyond the reach of physical science, such as the nature of the soul and the existence of God. However, there is no necessary connection between the construction of ontologies and pure, non-empirical philosophical speculation.

Ontologies are found across the full range of human inquiry, and their construction is a natural part of humanity's persistent effort to understand the world, whether by non-empirical speculation or hard-nosed experimental research. Natural science can be viewed as an example of classical ontology *par excellence*. Subatomic physics, for example, develops a taxonomy and corresponding theory of the most basic kinds of objects that exists in the natural world (electrons, protons, muons, and so forth). At the other end of the spectrum, astrophysics, for example, seeks to discover and characterize the range of objects that exist in its domain — quasars, black holes, gravity waves, and the like. Similarly, the life sciences categorize and describe the various kinds of living organisms that populate the planet. Of course, such examples can be multiplied from geology to psychology,

84

chemistry to sociolinguistics. Furthermore, this sort of inquiry is not limited to the natural sciences. The abstract sciences as well — mathematics, in particular — may be thought of as an attempt to discover and categorize the domain of abstract objects: prime numbers, transfinite ordinals, continuous nowhere-differentiable functions, and so on.

However, the natural and abstract worlds do not exhaust the applicable domains of ontology. Any well-designed product of human engineering and manufacturing — whether an automobile production plant or a semiconductor chip — must be as judiciously and exhaustively characterized by its designers as a new polymer is by its creators. The dramatic increase in the use and sophistication of computer technology, coupled with the need for greater collaboration in design, engineering, and manufacturing, has led to the idea of capturing complex engineering and manufacturing ontologies explicitly in large, sharable, reusable ontology knowledge bases. This leads to the modern notion of ontology arising from recent research knowledge sharing technology. With regard to content, an ontology in the modern sense is essentially identical to a traditional ontology: it is an account, a *theory*, of what exists in some domain. The difference is chiefly a matter of form; a modern ontology is couched in a standardized, computer processable language to facilitate storage and exchange of knowledge across computer systems.

Basic Conceptual Foundations: Kinds, Instances, Properties, and Relations

The central ontological concept is that of a *kind*. Historically, a kind is an objective category of objects — known as the *instances* of the kind — that are bound together by a common *nature*, a set of properties shared by all and only the members of the kind. For example, it is reasonable to say that the nature of gold is to have the particular atomic structure that it has: everything that has this property is gold, and everything that lacks it is not gold. Furthermore, in contrast to nonessential (or *accidental*) properties like *shape*, this property is essential to every instance of gold: no instance of gold could possibly lack it, otherwise, it would not be gold. Thus, by the traditional conception, to divide the world via an ontology is simply to identify the nature of each relevant kind in a given domain.

While there is an attempt to divide the world at its joints in the construction of enterprise ontologies, those divisions are not determined by the natures of things in the enterprise so much as by the roles those things are to play from some perspective or other within the enterprise. Because those roles might be filled in any of a number of ways by objects that differ in various ways, and because legitimate perspectives on a domain can vary widely, it is too restrictive to require that the instances of each identifiable kind in an enterprise share a common nature, let alone that the properties constituting that nature be essential to their bearers. Consequently, enterprise ontologies require a more flexible notion of kind. Toward this end, we must modify our terminology. To avoid overloading the term "nature," the properties associated with membership in a given enterprise kind **K** are known simply as its *defining properties*. Second, certain kinds of objects in large systems do not possess a clearly delineated set of associated

properties that are individually necessary and jointly sufficient for membership in the kind. Rather, for instance, domain experts might only "know them when they see them." Hence, it is not required that all defining properties of a kind K apply to all instances of K; rather, they are to be understood only as *characteristic* properties of Ks, properties that, in some combination or other, apply to most instances of the kind. (This can of course be strengthened in those cases where there are determinate, well-defined, necessary, and sufficient conditions for membership in a kind.)

An example will help illustrate how this conception of a kind is useful in the context of human-designed systems and will help clarify one way in which an ontology might function in information management. Consider the following representation of the basic ontology of a manufacturing cell composed of five entities: objects enter the cell and encounter a cutter, a drill, an inspection station, and two cleaners (Table 4).

Table 4. Characterization of a Manufacturing Cell

KINDS	DEFINING PROPERTIES
A: Cutter	{Diamond tool, ...}
B: Drill	{High speed motor,...}
C: Inspector	{High intensity lens light...}
D: Cleaner and Painter	{Dust filters, high gloss paint,...}
E: Cleaner	{Liquid cleaners,...}

Table 4 shows the kinds of objects that populate the system and lists the defining properties of each kind; one or two representative-defining properties are listed for each kind. Thus, the table provides an abstract representation of the *general* structure the manufacturing cell must exhibit at any given time. The property, **having a diamond cutting tool,** is a defining property of the kind **Cutter**. However, suppose the cutter which is instantiating this kind has the capacity to switch from diamond cutting tools to carbide. Even though **having a diamond cutting tool** is a *defining* property of the kind **Cutter**, it is nonetheless an *accidental* property of the cutter; it would lack the property if someone were to swap out the diamond tool for a carbide tool. The fact that it is a defining property of the kind means that, at any given time, whatever is playing the role of the cutter in the manufacturing cell has a diamond cutting tool, irrespective of whether the cutter that is playing the role has a diamond tool essentially or accidentally.

The point is that things can belong *contingently* to important kinds of objects in human-designed systems because the kinds in such a system are usually *artifacts*, human constructions. Hence, it is possible for an object of

one kind to "mutate" into an object of another kind simply by virtue of undergoing some nonessential change (e.g., the switching out of cutting tools). In other words, the broader notion of a kind is used because, when we build an ontology for a human-designed system, we are not setting out to discover and classify the world as it is in itself. Rather, we seek to divide up and categorize the objects in the system in useful and informative ways. An ontology's categorization scheme is justified only insofar as it is useful in organizing, managing, and representing information in the system. If objects of a certain kind K play a useful role in the system, we can justifiably admit them into the system's ontology, whether or not the defining properties of K are essential to its members.

The final basic ontological category is that of relations. Relations are general *connections*, or *associations*, between things. The relation **works-in**, for instance, is a general feature that holds between an individual and the department in which that individual works. Like a property, it is both *multiply instantiable* (i.e., different pairs of things can stand in the same relation) and *intensional* (i.e., a relation's identity does not consist in its instances).

The relations in an ontology are *typically* binary; that is, they hold between two entities, as with the relation **works-in**. However, there is no theoretical bound on the "arity" (number of arguments) of a relation; the relation **between**, for instance, holds between three objects. More artificial but nonetheless useful relations can easily be defined with four or more arguments. The Ontology Method, IDEF5, thus places no restriction on the arity of the relations that can be introduced into an ontology. Both properties and relations come in different *logical types*, depending on the kind of things to which they apply. Those that apply to other properties and relations — things other than individuals — are known as *higher-order* properties and relations, and are discussed in more detail below.

Motivations for Ontology

The ability to fix a domain vocabulary and its meaning in the context of use is critical for true CE. A large engineering or manufacturing project involves the resources of many different clusters of cooperative agents (human or otherwise) in the given endeavor. Each cluster makes its own contributions, and the overall success of the project depends largely on the degree of integration between those different clusters throughout the development process. Key to effective integration is the accessibility of ontologies characterizing each domain addressed by each cluster. For instance, access to a manufacturing ontology that includes constraints on how a given part is manufactured can aid designers by giving them insight into the manufacturing implications of their design concepts. Similarly, access to an engineering ontology that includes constraints on how a given part must function given a particular shape can aid process planners in their development of the appropriate manufacturing processes. Thus, a commonly accessible collection of relevant ontologies permits more efficient sharing of information from various sources in the enterprise.

A related motivation for constructing ontologies is to *fix terminology*. Two enormous problems in the coordination of large projects are the distributed nature of large enterprises — which makes face-to-face

communication particularly difficult to establish — and the diversity of backgrounds the various kinds of engineers bring to their respective roles. As a consequence, many managers, designers, and engineers in the same enterprise use technical terminology in different ways, with many different connotations. The problem is compounded in a virtual enterprise setting, where distributed agents must work together while dealing with potentially great differences, not only in technical background but also in corporate cultures. Because of such differences, the information one engineer *intends* to convey to another may become garbled; such miscommunications can result in lost time and resources. Consequently, it is often necessary in the course of a large project to fix the meanings of relevant vocabulary clearly and, when differences in usage cannot tolerated, standardize on one, unambiguous set of terms. Ontologies provide the means for fixing meanings and any necessary standardization..

A further strong motivation for ontology is *reusability*. Of the many problems in engineering and manufacturing, redundant effort expended in capturing information that has already been recorded elsewhere is one of the most significant. For example, in programming, the same kinds of routines (e.g., in the design of user interfaces) are often used in different programs by (in general) different programmers. Consequently, enormous amounts of time and effort have gone into reinventing the wheel. Recognition of this problem has led to the development of vast *libraries* that contain often-used routines which programmers can call into their programs rather than having to duplicate the function of existing code. Engineering and manufacturing face the same type of problem. For example, manufacturing domains share many features that are independent of the specific characteristics of a given domain. The more similar the domains, the more such features they share. Rather than encoding this information again in every new setting, analogous of the concept of a programming library, one can imagine collecting this common information into *ontology libraries* (i.e., large revisable ontology databases of structured, domain-specific information). Information in these ontologies can then be reused and modified to suit the needs of the moment. Moreover, because ontologies provide a standardized terminology by their very nature, no special additional effort need be expended on fixing domain terminology. Despite the potential size of a given ontology, ontologies are no less effective in smaller contexts than in very large ones.

There is a larger vision behind the idea of ontology development. Spearheaded by the Knowledge Sharing Effort sponsored by the Advanced Research Projects Agency (ARPA), ontologies are being constructed for a growing number of manufacturing, engineering, and scientific domains. With such ontologies in place, the advantages noted above could be realized on a global scale: standardized terminology with precise meanings that are fixed across industries and across international borders, and the ability to access and reuse a huge number of existing ontologies in the design and construction of new systems.

All these considerations strongly warrant the development of a well-defined ontology *method*. Previous approaches to ontology have been almost exclusively academic. Researchers from varied fields such as artificial intelligence, philosophy, database management, mathematics, and cognitive science have studied ontology from different perspectives. Hence, there has been little focus on a practical method for ontology acquisition. The

IDEF5 ontology description capture method developed under the IICE Ontology Thrust provides a mechanism to develop, store, and maintain scalable and reusable ontologies. The IDEF5 method assists domain experts and knowledge engineers in the construction of reusable ontologies. We have designed the IDEF5 technique to be usable by personnel of varied skill levels and from a variety of organizations.

The Role of Ontology in Enterprise Integration

Ontologies regarding CE specifically are directly relevant. Specifically, a serious roadblock to concurrent design and engineering is that the *data* generated in one enterprise context often cannot be *interpreted* in other contexts due to the lack of relevant background information. For example, the central engineering principles of one context might be unknown to another. However, if such background information is available in the form of ontologies, and if those ontologies are supported by appropriate tools, this roadblock can be overcome. That is, if the background information required for interpreting the data in other contexts is available, the intended *information* conveyed by the data can be extracted. A growing body of research is centering on the use of ontologies for meeting this challenge. In the context of the IICE project, the Experimental Tools Thrust of the IICE project — in particular, IDSE — presents a detailed architecture for information model integration that relies heavily on the construction of ontologies.

A key result of the Ontology Thrust was the characterization of the role of ontology in enterprise integration. An *information integrated system* relies primarily on the flow of information between agents situated in various parts of the system for the activation and control of the system. The critical factor in realizing such an information integrated system is this: *data received by a given situated agent must be interpreted correctly* (i.e., the information conveyed must be the information intended). In other words, information integrated systems must be concerned not only with the correctness of the *semantics* of the data — that is, its general context-independent meaning — but also with the correctness of the *interpretation* of that data. Interpretation in practical settings depends on the *context* of the recipient of the data. Correctness of interpretation implies an overlap in the knowledge bases of the agents communicating via the data being passed. Thus, for a system to be information integrated, there must be a careful mapping of the types of situations in which data will be conveyed and an alignment of the background knowledge of the situated agents who will be generating and interpreting that data in those situations.

The correctness of interpretation is the main factor driving the evolution of an enterprise towards an integrated whole. Without the background knowledge of the manufacturing context, designers cannot be certain of the implications of their design specifications or even how those specifications will be interpreted by the manufacturing personnel. Without background knowledge of the effect of a product (or its disposal) on the environment, designers cannot understand the environmental impact from their designs. Embracing a holistic view of the enterprise in its environment requires assimilation and alignment of these background knowledge assets, and

89

embodiment of the information system with that knowledge. The degree of information integration is measured by the amount of knowledge available throughout the enterprise to interpret correctly the data that connects the enterprise.

How Ontologies Support Different Flavors of Integration

The usefulness of ontologies is not limited to engineering and manufacturing information integration. Ontologies can be used as mechanisms to integrate many different kinds of repositories. Model integration is of major importance to enterprise integration initiatives. Models are idealized representations of the real world that are developed with a specific purpose. The integration of a set of models occurs chiefly through information sharing — the process of keeping each model apprised of new information which may enter any one of them. A set of models is truly integrated only when each model has access to relevant information in each of the others and is updated to reflect the most current state of that information. The most natural way to accomplish information sharing is to propagate new information around the models, allowing each to extract that part which directly influences its state. Thus, intermodel information transfer is the backbone of model integration. Unfortunately, transferring information between models is not always straightforward. A set of models, such as those generated by the IDEF methods, might express radically different kinds of information. Furthermore, different modeling methods typically do not provide exactly the same view of information, so the impact of a given statement in one model on another will probably be indirect. A straightforward one-to-one mapping will not always be available to move information across the models. Thus, the effect of a change in one domain model on the rest of the models of the domain cannot be computed using exclusively procedural knowledge and clear-cut, unambiguous algorithms. A more "intelligent" approach must be used. Consider the role of ontologies in supporting integration among a collection of models managed by an enterprise. Typically, these models will be of different varieties — data models, function models, process models, simulation models, and so on. Ontologies provide the "background" or context information that is crucial for integration among the models. The flow of information possible between the different models is generated using the information provided by the ontologies of the modeling methods and of the enterprise being modeled (Figure 20).

To accomplish model integration, we must first integrate the underlying methods. That is, we must rationalize the differences and standardize the similarities. *Method integration* is defined as integration among the concepts and implications of different modeling methods. Modeling methods have concepts and terms that have both overlapping and non-overlapping parts. Method ontologies help identify the relationship between the concepts and terms of the different methods with reference to the real-world concepts that the methods are designed to model. For example, an IDEFØ ontology will record that a function box represents a real-world activity; an IDEF3 ontology will state that a UOB box represents a real-world process. These two ontological assertions, along with the common sense knowledge that activities and processes are conceptually similar, lead us to infer that IDEFØ function boxes represent concepts that are similar to concepts represented by IDEF3 UOB boxes.

90
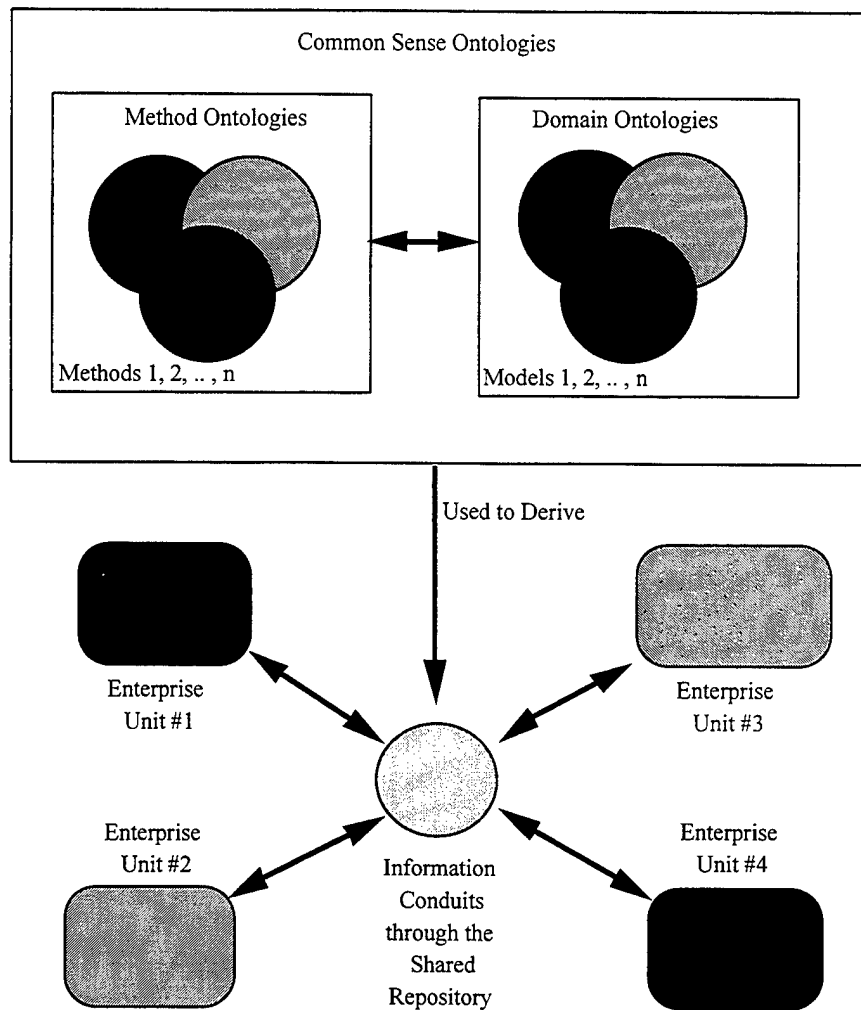
Figure 20

Role of Ontology in Enterprise Integration

Model integration is defined as the integration between the information contained in the different instances of models (the models need not be of the same type). In addition to the method ontology, two kinds of enterprise ontological knowledge play a role in model integration: domain-specific knowledge and common sense knowledge. An example of domain-specific knowledge is a business rule such as, "The purchase order needs to be authorized by the purchase manager." An example of common sense knowledge is the axiom, "An object cannot be at two different locations at the same time." Common sense knowledge is vital to different kinds of integration for an obvious reason: common sense knowledge is applicable across a wide range of domain situations.

The different kinds of ontologies provide the background knowledge required to enable the information flow that is necessary for model integration in a shared repository. This idea is illustrated in Figure 20. A more detailed description of the role of ontology in enterprise integration is given in, "The Role of Ontology in Enterprise Integration" (Mayer et al., 1993).

Accomplishments

The research and development approach has been to apply the basic ontology concepts to capture and represent actual domain ontologies. The results of the experimental ontology acquisition exercises helped develop insights about the nature of ontology and problems in practical ontology acquisition. These insights, in turn, helped identify conceptual, representational, and procedural enhancements to the IDEF5 Ontology Description Capture Method.

Summary of Domain Ontologies Developed

Ontology development was not an end in itself; rather, it was a means to accomplish the goals of the Ontology Thrust— that is, understanding the nature of ontologies and providing practical guidelines for ontology acquisition. The ontology development approach was to leverage off the results of ongoing ontology characterization efforts at KBSI and Texas A&M University. The application domains that were the focus of these projects served as a test-bed for the evolving IDEF5 method. Ontologies were developed in seven application areas:

1. Manufacturing process planning

2. Manufacturing process controller

3. Information modeling

4. Bill of materials

5. Shop floor control systems

6. Space mission planning

7. Simulation model design.

Manufacturing Process Planning. A process planning ontology was developed as part of a Defense Advanced Research Project Agency (DARPA, now ARPA) Phase I Small Business Innovative Research (SBIR) initiative at KBSI (deWitte et al., 1992). The evolving IDEF5 method was used to guide the ontology capture effort. The process planning ontology that resulted from this effort was used to design the architecture of a knowledge-based process planning system for DARPA. This exercise demonstrated the need to develop mechanisms to represent both static and dynamic (time-varying) knowledge as part of ontology acquisition. For example, we need to be able to represent taxonomies of processes and activities in a process planning ontology.

Manufacturing Process Controller. A manufacturing process controller ontology was developed under a DARPA-funded Phase I SBIR project. The process controller ontology that resulted from this effort was used to design the architecture of a knowledge-based shop floor controller. This development effort helped us identify the

need to provide mechanisms to represent object states at multiple levels of abstraction. This requirement led to the incorporation of nested state diagrams in the IDEF5 method.

Information Modeling. Theoretical foundations for method integration were developed in a National Science Foundation (NSF) SBIR Phase I project. In addition, the Experimental Tools Thrust has developed ontologies of the different IDEF methods and mappings between the methods to facilitate automated inter-method translation. The resulting suite of method ontology libraries will provide key background knowledge for enabling automated method integration. Some of the insights gained as part of analyzing method ontologies were published in "The Role of Ontology in Enterprise Integration" (Mayer et al., 1993). The method ontology acquisition efforts led to the identification of important enhancements to the IDEF5 Elaboration Language.

Bill of Materials (BOM). A Texas A&M University project funded by the Digital Equipment Corporation led to the development of a BOM ontology for manufactured parts. IDEF5 was used to structure and guide this ontology capture effort. This ontology acquisition and analysis exercise helped demonstrate the need for expressing knowledge about connections between associations. Such connections are modeled as (second-order) relations in IDEF5. The results of this effort were also used to refine the library of meronymic (i.e., *part-of*) relations in the IDEF5 relation library.

Shop Floor Control Systems. A shop floor control system ontology was developed as part of a Texas A&M University project. The resulting ontology helped develop the architecture of a knowledge-based shop floor control system. This development effort helped identify the need for practical guidelines to assist in managing complexity in ontology. This led to the design of different kinds of decomposition mechanisms in the IDEF5 Schematic Language.

Space Mission Planning. KBSI is developing an ontology of Space Mission Planning as part of a Phase II National Aeronautics and Space Administration (NASA) SBIR project. The Mission Planning ontology will be used as the basis for designing and building a knowledge-based planning and scheduling system for NASA. The results of our ontology analysis at NASA helped identify fine-grained ontologies of time. The resulting temporal knowledge characterizations have been encapsulated in the IDEF5 Relation Library.

Simulation Model Design. KBSI developed an ontology of simulation model design as part of an NSF Phase II SBIR project. This ontology assisted in the development and implementation of a commercial knowledge-based simulation model design tool, PROSIM™. The simulation ontology acquisition effort helped to harmonize the "object-centered" and "process-centered" world views. This harmonization led to the incorporation of the object state schematics as part of the IDEF5 Graphical Language. Moreover, it helped us identify improvements to the IDEF3 method.

Theoretical Developments

The Formal Framework of Ontology. Two related matters are of the highest importance in the design of any system of representation. First, the *type* of information that is to be represent must be considered. That is, one must determine the basic semantic categories needed to capture the information in question (e.g., classes, relations, events, etc.). Second, one must consider the *expressive power* necessary to express the information that will be required to meet one's representational needs.

The need for greater expressive power is met in the theoretical foundations of IDEF5 by first imbuing its underlying formal knowledge representation language with the full power of first-order modal logic (Hughes & Cresswell, 1968). The underlying knowledge representation language will be based on the same situation theoretic language that serves as the foundation for IDEF3 and the IST Thrust, allowing for straightforward sharing and integration of relevant information.

The power of first-order logic is well known and greatly exceeds the expressive power of information modeling methods such as IDEF1. Modal logic extends first-order logic by introducing operators for necessity and possibility, and a corresponding set of theoretic semantics. This extension, among other things, allows for a natural expression of facts about essential and accidental properties. (An essential property of x is a property that x must have in order to exist — that is, a property that is *not possible* for x to lack.)

The standard set of theoretic semantics for modal logic is discussed in terms of the heuristic concept of a "possible world." The idea goes back to the philosopher/mathematician Leibniz who proposed that the world could be other than the way it is. These "ways the world could be" may be thought of as other possible worlds. Of course, one way the world could be is the way it actually is. Thus, the *actual world* is one of the possible worlds.

It is illuminating to think of systems in terms of possible worlds. In importing the ontology enterprise into the information modeling domain, the concern is not with the world *per se* but with the world of an organized system. Accordingly, in this context, possible worlds should be thought of as alternative states of the system. Thus, a relational database model could be thought of as modeling all the possible states of the database (i.e., all the different possible relations that could populate the database). Thinking in these terms often helps one design more breadth and flexibility into the model in anticipation of possible, but unlikely, states. In addition to providing a definition of the notion of essential and accidental properties, the "possible worlds" concept helps one anticipate or consider all possible natural kinds that might appear in the system and thus define a sufficiently broad ontology.

The framework of possible worlds is also the informal framework chosen by the members of the International Standards Organization (ISO) working group to characterize the notion of a conceptual schema (see the Three-Schema Architecture Thrust section of this document). A conceptual schema consists of all the *necessary*

propositions that hold in a given system in all possible worlds or in all possible states of the system. Our use of the framework here ties in naturally with our work on the development of the three-schema architecture.

Higher-Order Properties and Relations. In addition to modal logic, *higher-order properties and relations* are required to represent a given ontology properly. Intuitively, properties and individuals are of different logical types. Properties are the abstract, general features that are shared by distinct individuals; that is, the *ways* in which distinct individuals can be the same. Similarly, relations are the general associations which meaningfully can be said to be shared by distinct pairs (triples, etc.) of individuals — for example, the property of having mass or the *part-of* relation that holds between a concrete thing and its parts. Thus, properties and relations are identified by abstracting away from the particular features of individuals and, hence, are often characterized as being of a *higher* (i.e., roughly, more abstract) logical type than the individuals that exemplify them. Thus, individuals are frequently referred to as *first-order objects*; properties and relations of first-order objects are referred to as *first-order properties and relations*. However, properties and relations that hold among individuals are identifiable (albeit abstract) entities themselves. As such, they can have properties that apply to them but not to individuals (e.g., the property *having at least one instance*). Such higher-order properties are typically known as *second-order properties* because they apply to first-order properties and relations rather than first-order individuals. There are interesting second-order relations as well. For example, the relation *has-more-instances-than* is a relation that holds between two kinds. Again, the *subkind* relation is a relation that holds between a given kind and a more general kind that subsumes it (e.g., the kinds *human* and *mammal*, or *NC machine* and *machine*). However, some second-order relations include individuals among their arguments. For example, the *instance-of* relation holds between an individual **a** and a kind **K** just in case **a** is an instance of **K**. Such "mixed-type" relations that hold between objects of different logical types are nonetheless second-order. Therefore, a second-order relation is a relation that always includes at least one first-order property or relation among its arguments.

In summary, the IDEF5 ontology includes three logical types: individuals (or first-order objects), first-order properties and relations that apply to those, and second-order properties of and relations that apply to at least one first-order property or relation.

Process Kinds and States. An adequate characterization of the kinds that inhabit a given domain often cannot be divorced from the *processes* in which their instances are involved. Typically, processes involve two sorts of change: change in *kind* and change in *state*. For example, in an incineration process some quantity of wood is transformed into ashes and gas; the wood is destroyed and quantities of ash and gas result. By contrast, the process of ice melting simply involves a change in the state of a given quantity of water from frozen to liquid; the water itself is not destroyed, only altered in a nonessential way. This is in fact what generally distinguishes states from kinds: unlike kinds, states are usually *contingent* groupings in a domain. The distinguishing feature of a state is usually a changeable, accidental feature of a thing, such as water being *frozen* or a car body being *painted*. Both sorts of changes are accommodated in the IDEF5 ontology capture method.

Like individual objects, processes also cluster naturally into general categories. For instance, temporally distinct events in which a manufacturing process plan is generated from a given design are all instances of the general process of manufacturing process planning. Thus, general processes, like the kinds discussed in the previous paragraph, are multiply instantiable; distinct, individual events can be instances of the same general process. Furthermore, the identity of a general process is independent of its instances; it remains the same regardless of whether or how it is instantiated. Hence, like kinds, a general process is intensional. Therefore, general processes can be thought of as kinds no less than object kinds. However, unlike the instances of individual kinds, processes are *things that happen*. Thus, not only do they "contain" other objects as parts (like the instances of complex kinds), they *occur* over an interval of time, and things are *true of* the objects in the process during at least some parts of that interval. It is this fact that often makes it necessary to refer to relevant processes in the characterization of a given kind.

Because of the importance process kinds can have in the definition of a domain ontology, IDEF5 permits one to refer to them no less than object kinds. However, there are two distinct contexts in which such reference can occur, and the information that is kept about a process kind will differ depending on the context. If a process kind **P** is referred to in the description of a transformation or transition involving two kinds of objects, the "internal" character of **P** is described in accordance with the IDEF3 process description capture method. That is, **P** is described in terms of the object kinds it involves, their properties, and the relevant relations that hold between instances of those kinds when the process in question is instantiated. In particular, in such contexts, the usual sort of information kept about an object kind — its defining properties and so on — is not kept about the process kind.

On the other hand, to understand a domain it may be important not only to know how objects are involved in the internal structure of a process, but also — as with object kinds generally — how one kind of process relates logically to another kind of process, independent of the details of its internal structure. For instance, *manufacturing process planning* is a subkind of *planning*. In these cases, process kinds are characterized exactly like object kinds; that is, defining properties are identified and so on. Two distinct constructs are provided in the IDEF5 graphical language corresponding to these two possible characterizations of process kinds.

A Catalog of Generic Relations. The ontology effort developed a knowledge-rich repository composed of a set of definitions and characterizations of commonly used relations. This knowledge catalog, called the IDEF5 Relation Library (IRL) was motivated by an important observation: knowledge about the world can be classified, based on the level of specificity, as *common sense knowledge* and *domain-specific knowledge*. Common sense knowledge about real-world associations (relations) is encapsulated in the IRL. To understand further the motivations for the IRL, consider the field of software engineering. Often in software development, the same kinds of routines are used in different programs by (in general) different programmers. In earlier times, tremendous amounts of time and effort were lost for lack of the ability to reuse work. Recognition of this problem has led to the development of extensive *libraries* that contain frequently used routines which programmers can simply call into their programs. Such libraries have eliminated the need to duplicate the functionality of existing code. The

development of ontologies will face the same sort of problem (and solution). The same or similar relations will likely appear in a number of different ontologies.

The IRL is a repository of formally defined and characterized relations that can be reused and customized. The IRL will enable ontology developers to reuse and customize relations that have been defined in previously captured ontologies. The library can also be used as a reference for the different ways to define and characterize relations and illustrative examples of the use of the IDEF5 elaboration language. All definitions and characterizing axioms in the library are written using the IDEF5 elaboration language. The library is extensible in that any relation that has been formally defined and characterized may be added to it.

The IDEF5 library relations are grouped into the following seven categories:

1. Classification Relations (including class inclusion relations)
2. Meronymic Relations
3. Temporal Relations
4. Spatial Relations
5. Influence Relations
6. Dependency Relations
7. Case Relations

These relations are described in detail in the *IDEF5 Ontology Capture Method Report* (Mayer et al., 1994).

Summary of Ontology Development Insights

The ontology development activities resulted in a better understanding of the nature of ontology and provided useful insights for overcoming practical barriers to ontology acquisition. The insights developed during this period can be classified as follows:[10]

1. Better understanding of the conceptual and theoretical foundations of ontologies and the expressive requirements of a language for expressing ontologies.

2. Enhanced characterization of higher-order properties and relations.

3. Incorporation of process kinds in IDEF5.

4. Identification of a catalog of generic relations (i.e., IRL).

5. A rigorous account of the role of ontology in enterprise integration in support of CE.

_____

[10] Many findings of the Ontology Thrust are documented in the IDEF5 Method Report (KBSI, 1994).

This page intentionally left blank.

INTEGRATED SYSTEMS THEORY THRUST

This section presents a brief introduction to the Integrated Systems Theory Thrust and a list of the major accomplishments of the thrust.

Introduction

The goal of the IST Thrust is the development of an empirically well-grounded, mathematically sound theory of system integration. The purpose of such a theory is threefold:

- To provide an account of the nature of integration and integrated systems and, thereby, the means for explaining the phenomena of integration (e.g., what occurs among the agents and other objects in an integrated system).

- To provide a foundation for abstracting general principles of integration from the phenomena of integration.

- To provide a framework for developing an engineering science of integration, in particular, to identify the limitations of integration, to provide a framework for gauging the degree of integration in a system, and to develop a formal basis for developing methods and tools to support the integration of system models.

A theory with these aspirations must be both intuitively and formally sound. Intuitive soundness is important because, to be relevant, the theory must be driven by well-founded, widely shared intuitions about the nature and character of integration. Consequently, the IST has been developed using a sizable body of "phenomenological data" (observations) on the characteristics of integration; therefore, it is empirically well-grounded.

Formal soundness is important insofar as it ensures the theory is theoretically well-grounded. Essentially, two components provide the theory with its formal soundness. First, the theory is formulated in a precise, sufficiently powerful mathematical framework (i.e., a framework with the expressive capacities needed to represent the concepts and phenomena of integration in a robust manner). Second, the concepts and phenomena in the theory receive precise formal expression within the given mathematical framework. Analysis of the data has led to two related but distinct components of the IST: a *model* integration component and a *systems* integration component. The overall framework for both components is the NIRS, a formal system based on first-order logic and set theory.

The IST is fundamental to the entire IICE project. In particular, the NIRS has been used as the framework for a general theory of system integration, consisting of such basic notions as those of information flow, constraints, utility, and degree of system integration. Because of the dynamic, highly structured nature of complex systems, the framework for the system integration component must be capable of representing structured, dynamic information.

The NIRS was used to develop a version of *situation theory* to serve as the conceptual foundation of the theory of system integration. Additionally, the NIRS framework was used as the foundation for the development of the methods and techniques for enhancing integration among system models to support CE in large systems. Specifically, the NIRS was used to give the formal specification of the methods in the Methods Engineering Thrust and the Ontology Thrust; the account of model integration developed using the NIRS provides the theoretical foundation of Levels 1, 2, and 3 of the IDSE project in the Experimental Tools Thrust.

Significant Accomplishments

The most significant accomplishments of the IST Thrust are as follows:

- Collection and analysis of research and data on the central characteristics of integration.

- Development of the NIRS, a mathematical framework for the formal specification of the syntax and semantics of representation languages (modeling languages, in particular), as well as integration mechanisms between them. (The NIRS is described in detail in Appendix D of this document.)

- Development of a formal model of information sharing between knowledge bases expressed in heterogeneous representation languages.

- Development of a formal, situation theoretic framework to represent dynamic, as well as static, information. (See Appendix E for a detailed discussion.)

- Development of a formal, situation-based theory of integration and integrated systems.

- Participation in the International Conference on Enterprise Integration Modeling Technology (ICEIMT) sponsored by the United States Air Force (USAF) Manufacturing Technology Directorate (ManTech) at Wright-Patterson AFB and ESPRIT, the industrial consortium of the European Community. Dr. Christopher Menzel was invited to present a paper at Workshop I of the conference on the topic of Model Integration; this paper was among those selected to appear in the Massachusetts Institute of Technology (MIT) Press proceedings volume (Menzel, Mayer, and Sanders, 1992).

- Participation in major standards efforts directed toward information integration, including the Product Data Exchange using Step (PDES) Dictionary and Methodology Committee (responsible for the SUMM), and the IEEE IDEF∅ Formalization Working Group. Dr. Christopher Menzel and Boeing's Dr. Jim Fulton (chief author of the PDES SUMM) are the chief authors of the formalization. This formalization will be adopted officially as a national standard by the National Institute of Standards and Technology (NIST), IEEE and the IDEF Users Group, and will be submitted as the international

On the Foundations of an Integrated Systems Theory: Two Types of Theory

A necessary condition for the development of any formal theory is that one have a relatively clear understanding of the intuitive subject matter that motivates and informs one's theorizing. Theory then extends this subject matter in a deeper, more systematic way. As the theory develops, understanding of the phenomena increases, and new phenomena present themselves as objects for further research. For example, the intuitive subject matter of number theory shows itself in the phenomena of counting, grouping, separating, and so on. The theory leads to greater insight and provides material for further abstraction, such as the concept of a "group." In the 1930s, mathematicians sought a precise theoretical account of the intuitive notion — the phenomenon — of an *effective procedure*, or *algorithm*, for calculating a given function. Several independent characterizations arose — turing machine computable functions, lambda computable functions, and recursive functions, among the earliest and most prominent —all of which, remarkably, proved equivalent. The resulting formal theory of computability has grown into the field of theoretical computer science, one of the most fruitful and important branches of contemporary mathematical research.

The proper way to proceed in the development of an IST is to circumscribe the phenomena that motivate the development of such a theory; namely, the phenomena of integration. The examples above might suggest that the development of an IST will be able to proceed in much the same manner as the development of Newtonian mechanics or number theory. However, we can distinguish two possible cases one might encounter in the development of theory. Consider again the development of computability theory. The intuitive notion of an algorithm is quite clear: an algorithm is a step-by-step procedure for moving from a given input to a given output in a wholly determinate, repeatable fashion. Thus, a candidate formalization of the notion, a formal theory of computability, can be deemed successful insofar as all those functions (and only those functions) are calculable by an intuitively acceptable algorithm and are calculable according to the formalization. Of course, there is no way to know for certain whether any given formalization meets this test because there are infinite functions, calculable by algorithms, that we shall never test. The best we can hope for is corroboration, which in fact has been the result: in every case checked, a function calculable by an algorithm has turned out to be computable by a Turing machine. Consequently, a good deal of trust has been placed in these formalizations. This trust is codified officially in what is known as *Church's Thesis:* the thesis that algorithmic calculability and computability by a Turing machine coincide or, more generally, that the intuitive subject matter of algorithmic calculability is exactly captured by our formal notion of computability.

However, computability theory marks a rather spectacular success in the history of theory building. Not all theories have been so charmed. The success of computability theory is in part a function of the *clarity* of its

intuitive subject matter; though it is not trivial to characterize the notion of an algorithm precisely, there is generally a broad agreement when one hits upon its central components (e.g., the discrete, stepwise character of an algorithm). Hence, it is relatively safe to put faith in Church's Thesis. By contrast, consider the development of contemporary syntactic theory. Here the subject matter is *language*, in particular the notion of a *grammatical sentence* of a language. But there is far less agreement about the notion of grammaticality — in particular, about what counts as a grammatical sentence —than about the notion of an algorithm. Consider Chomsky's well-known example: "Colorless green ideas sleep furiously." Is this a grammatical sentence? It appears to have precisely the same form as, for example, "Penniless little children sleep peacefully." Thus, one might conjecture that the sentence is indeed grammatical but *semantically* ill-formed. On the other hand, grammar is supposed to separate "good" strings of words from "bad" ones; surely the above counts as bad in some strongly intuitive sense. So what is the truth of the matter? Is it, or is it not, grammatical?

The vagueness and complexity involved in the subject matter of a given theoretical domain is inversely proportional to the degree of certainty and unanimity there will be in regard to whether a given theory has captured the subject matter accurately. This has certainly proven true in the case of syntactic theory. A variety of linguistic theories have appeared since Chomsky's pioneering work in transformational grammar. However, unlike the theory of computability, these theories are by no means equivalent; they disagree in regard to their subject matter (i.e., in regard to the nature and extent of grammaticality) as well as in their overall theoretical structure. Again, in contrast to computability theory, all existing syntactic theories conflict at points with our intuitions about grammaticality. The best we can hope for is a *future* theory that exceeds all others in theoretical power and agrees with our intuitions at every point, at which time an analog of Church's Thesis might be appropriate.

The point is that the notion of integration is more similar to the notion of grammaticality than the notion of computability. Like grammaticality, most people have fairly clear intuitions about what integration consists of. However, the *precise* nature of integration and the nature of the information to be integrated — exactly what it consists of and how it is to be represented — is by no means obvious; unlike the theory of computability, the characterizations of equally informed experts will likely not prove equivalent. Far more probable, as in syntactic theory, some accounts will do some things better than others; these discrepancies will point the way to the construction of theories that more and more adequately meet our empirical needs. The development of the theory will be tentative and incremental as researchers come to a consensus on an intuitively and empirically acceptable notion of integration and on the nature of the information to be integrated.

Proper Representation of Information

An integral part of the development of an IST involves analyzing the nature of information across a wide range of system types, and developing an adequate representational medium for that information. One cannot reasonably hope for a theory of information integration without a rich and powerful theory of information. Recent

theories of information cast information as a genuine part of the physical world, as real, quantifiable, and subject to analysis as electrons and black holes. This view is crucial for an IST because, system information to be integrated, the information in question must be theoretically tractable. The question then is: what is the nature of this information, and how is it to be represented?

Currently, there are a number of information theories — the most widely known being perhaps that of Shannon and Weaver (1949). The major application of the theory has been in the theory of communication. Its most important area of application has been in the transmission of coded information (e.g., across phone lines) and especially in the preservation of accuracy in the presence of noise and other types of resistance that arise from the physical properties of the channels of communication.

A more recent theory of *algorithmic* information was developed by Chaitin (Chaitin, 1987). Briefly, algorithmic information theory studies the amount of information needed to convey a given piece of information, where the *amount of information needed* is the length in bits of the smallest program (instructions + data) required to convey the information in question.[11]

Algorithmic information theory is perhaps more appropriate in the context of theory of information integration than the theory of Shannon and Weaver because its explicit subject matter is information represented in computationally tractable forms. However, the chief difficulty with both theories is that they are essentially *bit level* theories of information; that is, the fundamental elements of information are unstructured bits. While in a certain precise sense all digitally stored information is bit-level information, we generally do not want this information *presented to us* at that level. The *structure* of information represented at the bit level is too suppressed, too deeply hidden within some particular coding scheme. Consequently, we need a theory that can represent information explicitly in a way that captures the higher level of organization and structure found in typical large human-designed systems and that can be understood and manipulated by ordinary, suitably trained human agents. Such a theory must represent information in a way that corresponds more directly to the way human agents represent information to themselves.

---

[11] For instance, it takes $\log 2\ n + c$ bits of information to convey (i.e., to generate) the information represented by a bit string of n ones — $\log 2\ n$ being the length of the binary representation of n, and c, a constant representing the length of a program for unpacking binary representations into corresponding unary strings. For values of n significantly larger than c, the information it takes to generate a string of n 1s is much less than n. By contrast, it can be shown that most strings are random in the sense that the amount of information needed to generate them is approximately equal to the length of the original string.

This being the case, *situation theory*, built on the rich and well-understood foundation of first-order logic, provides a more appropriate theoretical starting point (Barwise, Gawon, Plotkin, & Tutiya, 1991; Devlin, 1991). Situation theory is a recent, powerful theory of information and information transfer between situated agents— human or otherwise. Its original, and still immensely productive, application was in the area of natural language semantics under the name of "situation semantics" (Barwise & Perry, 1983). Because of the theory's original semantical thrust, it was fitted with a representational apparatus rich enough to distinguish subtle shades of meaning that permeate ordinary natural language usage. This characteristic gives situation theory numerous advantages over previous theories of information relative to the goals of the IST. Unlike bit-level theories of information, situation theory represents information in a way that captures the higher level of organization and structure found in models of typical large human-designed systems. Thus, it represents information in a form that corresponds more directly to the ways in which human agents represent information to themselves. The theory's basic structure includes those characteristics of familiar first-order logic: objects, properties, relations, and spatio-temporal intervals. These constructs, in turn, are the building blocks of intuitive, fine-grained units of information known as *infons*. Operations that yield logically complex infons give rise to a full-blown algebra of information that is tailored specifically to the needs of model integration. This rich store of structured information objects allows for elegant representations of a variety of phenomena, notably the following:

1. The distinction between general, or type-level, information (e.g., a general database design) and specific, or token-level, information (e.g., a specific populated database that realizes the design). Notably, the situation theoretic characterization of information flow between agents turns on this distinction.

2. Temporal and causal relations between dynamic events and processes.

3. The partial nature of individual pieces of information in a closed system; most states of affairs in a given system carry information about only a restricted portion of the system.

4. The (typically) incomplete nature of an agent's knowledge of a given system; models, in particular, are built and refined over time as knowledge of the system increases and errors are eliminated.

5. An intuitive account of the information *carried by* a given model and the *influence* of one model upon another.

First-order logic and situation theory provide a sound basis for developing a representational medium possessing both expressive power significantly greater than that of other approaches and constructs that provide natural representations of all entities that constitute the information one is likely to encounter in any feasible system.

The Central Characteristics of Integration

"Information integration" is a buzz word in industry and academia today, and research in this area abounds. Our intention in this section is to isolate the dominant characteristics of the notion from this research. The development of an integrated systems theory initially will consist of compiling and analyzing these data. Appendix C summarizes the state-of-the-art research in the field. The basic themes that emerge are the following:

1. Sharing of Common Data. Integration implies that common data can be shared between different tasks and applications. The most important aspect of this characteristic is that data should be entered only once, that only stringently controlled duplication of data is allowed, and that only organizational policy (as opposed to technological barriers) restrict access to the data.

2. Work Flow Support. A basic assumption of an integrated system is that it possesses (or is built around) an accurate model of the organization it supports. In fact, some of the earliest uses of the term "integrated system" referred to manufacturing systems where the physical organization of the facility and the design of the material-handling system were designed to fit closely with the process flow of the product. Integration supports the flow of work by allowing information arising in one stage of the development process to be accessible to team members working on another stage of the process.

3. Change Propagation Support. In a manner similar to work flow support, integration also implies support for change propagation. By maintaining relationships between data artifacts (and by having an accurate and usable internal model of change semantics), modification to one artifact can be propagated to other artifacts.

4. Flexibility with Respect to Change. An integrated system is normally assumed to have a degree of flexibility associated with it. This characteristic of flexibility is a measure of how easily the system can be expanded, contracted, or evolved to respond to new requirements. The degree of flexibility an integrated system exhibits will usually determine the life-span and scope of application of that system design.

5. Access to Global Environment. A characteristic of information model integration is to provide access to the global environment and to provide a global vision of how information flows in and out of an agent.

6. Data and Information Integration. Data and information integration is achieved when the right information can be used at the appropriate place and appropriate time whenever it is stored in the system under any format.

7. System Integration through Information Integration. Even if its information system is integrated, an enterprise may not necessarily be integrated. An information-integrated system, achieves integration through information integration. Such a system is driven by a network of integrated models, knowledge bases, and databases about its various parts, and it uses that information to support the flow of work. An important advantage of an information-integrated system is that it provides a natural framework for CE and Total Quality Management (TQM).

8. Feedback and Consistency. An "integrating system" should be capable of providing feedback from the enterprise to the models and the transformation information necessary for rendering the models and the enterprise consistent.

Theoretical Foundations: Neutral Information Representation Scheme (NIRS)

The core mechanism that we have settled upon for realizing the above integration architecture is NIRS. In accordance with the integration architecture sketched above, NIRS is designed to provide an overarching theoretical vantage point, independent of any particular modeling method, from which one can represent (1) the information expressed in any given model, (2) the way such a model's syntax and semantics enable it to express that information, and (3) relevant background information and constraints. Thus, NIRS provides the framework not only for a theory of integrated systems but also for the design and development of any corresponding model integration support tools.

Modern enterprises rely heavily upon enterprise *models*, that is, function models of business activities, database models, computer-aided design (CAD) product models, dynamic models of manufacturing processes, and so on. These models come in a wide variety, ranging from "box-and-arrow" graphical languages to sophisticated CAD packages to full-blown programming languages. The central insight behind this proliferation of modeling methods is that a given enterprise typically contains many different kinds of information — the relatively static information such as that which might be stored in an employee database, the dynamic information involved in planning a manufacturing process, the structured information employed in product design and so forth. Consequently, a method tailored to one type may be quite unsuitable for another.

However, this proliferation of modeling methods has led to a new problem. For a complex manufacturing or engineering environment to be effective, the activities in that environment must be tightly coordinated. This coordination is possible only if consistency across models is efficiently maintained and the information carried by one model is accurately and efficiently propagated to other models that manage related information. These two aspects of model integration are called, respectively, the problem of *intermodel consistency* and the problem of *information propagation*. Unfortunately, most methods on which the majority of enterprise models are based were

106

developed independently of other methods. Hence, there is no general theoretical foundation for the needed integration, and integrating the models in a complex environment typically amounts to manual translation by means of ungeneralizable rules. The challenge of enterprise model integration is to provide a general foundation and bring it to fruition in the form of usable integration methods and tools.

One approach to this problem is to develop a *global representational medium* capable of representing, in a uniform way, the information contained in any possible information model (and in any database constructed from any such model). With such a medium, one could express the *informational links* that exist between enterprise models. These information links could form the basis for automating information transfer across models in a timely fashion. This would be a truly integrated environment.

NIRS is a global representational medium based on the notion of a *first-order language*. First-order languages are flexible, expressively rich, and well understood. They are used widely in mathematics, linguistics, philosophy, and computer science wherever clarity of expression is especially important. Many familiar mathematical theories (e.g., the theory of sets, Boolean algebra, topology, etc.) can be expressed in first-order terms. More recently, first-order languages have found their way into the domain of artificial intelligence (AI) where first-order languages find straightforward representation in AI programming languages like LISP and PROLOG (PROgramming in LOGic). Indeed, first-order mathematical logic is the formal foundation of PROLOG (Clocksin & Mellish, 1984).

Generally speaking, a first-order language L is a *formal* language. That is, it is a formal structure consisting of a fixed set of basic symbols (often called the *vocabulary* of L) and a precise set of syntactic rules (its *grammar*) for building up the proper sentences, or *formulas*, of the language that are capable of bearing information.

The syntax, semantics, proof theory, and temporal and modal logic of NIRS are included in Appendix D. In the next section we provide an informal overview of our adaptation of situation theory sufficient for presenting the central developments in the IST. In Appendix E, we provide a more formal account of the theory.

Situation Theory

The basis of our work on the foundations of the representation of process information is *situation theory*, a relatively recent, highly expressive theory of information (Barwise & Perry, 1983; Barwise, 1989; and Devlin, 1991). In this theory, situations are (typically) concrete, spatially and temporally extended pieces of the real world, such as a baseball game, a math class, a manufacturing system. However, they can also include other situations as well as nonconcrete systems like the field of real numbers.

Basic Infons

A situation is individuated by the pieces of information it *supports* or *holds*. In situation theory, individual pieces of information are known as *infons*. Infons are structured; that is, in a given domain they are composed of objects, properties, and relations that exist in the domain. (Objects here are construed broadly to include not only physical objects but also abstract ones like numbers and temporal intervals, as well as other situations). More specifically, the *basic* infons in a given situation s are the *fundamental units of information*, good and bad, "generated" combinatorially from the relations and appropriate arguments for those relations within s; that is, the basic infons of s consist of all possible legitimate units of information of the form

objects $a_1, \ldots, a_n$ stand in relation r,

and

objects $a_1, \ldots, a_n$ do not stand in relation r,

where r and the a_i are all constituents of s. We represent these infons as $\langle r, a_1, \ldots, a_n, + \rangle$ and $\langle r, a_1, \ldots, a_n, - \rangle$, respectively. A basic "positive" infon $\langle r, a_1, \ldots, a_n, + \rangle$ holds in s just in case its component objects $a_1, \ldots, a_n$ stand in the relation r in S, and a basic "negative" infon $\langle r, a_1, \ldots, a_n, - \rangle$ holds in S just in case $a_1, \ldots, a_n$ are present in s and are appropriate for r in s but do not stand in that relation in s. Thus, for example, the infons $\langle$**mother-of**, Hillary, Chelsea, $+\rangle$ and $\langle$**mother-of**, Chelsea, Hillary, $-\rangle$ hold in typical White House situations in 1993; by contrast, $\langle$**mother-of**, Hillary, Chelsea, $-\rangle$ *fails* in such situations.

Infons that fail in certain situations account for *mis*information — information which, while carried by a certain situation (at least ostensibly), nonetheless does not hold in it. For example, an alarm clock going off at 6:30 a.m. on a Saturday morning might convey to its groggy owner the misinformation that it is time to get up for work. Such a capacity to account for misinformation is crucial to a full-blown theory of information because it is often important to specify, not only the information that *holds* in a given situation but also the information that fails in it. System errors, for example, are often the product of miscommunication (i.e., the transfer of misinformation from one element within the system to another). The cause of such miscommunication is typically a false "belief" on the part of some (human or mechanical) system element. In such a situation, the task of a system analyst is to isolate the source of misinformation.

Because situations are, in general, *limited* pieces of the world, an object **a** that exists in one situation s may not exist in another s'. Hence, s' will be "silent" on **a**; more precisely, it will support no information about **a**. That is, situations, are *partial with respect to information*; they do not answer every question about every individual or every state of affairs. For example, a typical baseball game in the Houston Astrodome carries no information about the price of Coho salmon in Seattle's Pike Place Market. It is as important for a theory of information to be able to

represent partiality as it is to represent misinformation. System error and inefficiency arise not only through misinformation; they can also occur through a *lack* of information. The ability to represent partiality enables an analyst or engineer to chart the informational content of various aspects of a given system and thereby isolate informational gaps that might be affecting the system adversely. This adaptation of situation theory has captured partiality by declaring an infon to be *undefined* with respect to a situation which is "silent" about it.

Complex Infons

As the name suggests, basic infons are not the only infons — information does not come only in simple structures. Rather, pieces of information can "combine" in well-defined ways to form more complex infons. More formally, the domain of infons forms an *algebra*, a structured set that is closed under various operations. These operations correspond roughly to standard binary logical operations in predicate logic — conjunction and disjunction — as well as the operations of existential and universal quantification. These operations are described in more detail in Appendix E.

Situation Types and Object Types

Two broad classes of process information exist:[12] *type-level* information and *token-* (or *instance-*) *level* information. Type-level information has to do with the *general* structure of a given situation and is the typical level of information represented by, for example, an entity relationship (ER) database schema model, an IDEF3 process model, or an informal process description. An IDEF3 diagram of an automobile assembly line, for example, expresses what happens *in general* to car bodies, engines, and so forth as they move through the system, not what happens to a *particular* car body or a *particular* engine. This is especially evident in the case of models of unimplemented processes, such as process models that arise during the design phase of a given product. In such cases, modeling entities can represent only the general entities and processes conceived by the designers because those entities and processes will not exist until the design reaches production.

However, it is important to represent instance-level information as well, especially for the purpose of *activating* a process model. For example, it is highly desirable to be able to test the structure captured in a model by observing the behavior of several instances of that structure. Once the model accuracy has been determined, the model can be used to record instance-level information. For example, a system analyst might record instance-level information about the progress of car bodies through an assembly line to determine why the actual jobs per hour of the system fall short of what is predicted by the type-level process model. Moreover, relevant information is not exclusively at the type level or at the instance level but may involve both levels at once. The instance-level

---

[12] Of course, other kinds of information exist as well.

information garnered by the system analyst, for example, may have implications for the general type-level structure of the manufacturing process being analyzed (i.e., in the type-level process model that stores that structure).

Type-level information and the correspondence between tokens and their types are represented in situation theory in a variety of ways. The most important representation of type-level information comes in the form of *object types* and *situation types*. Object types are full-fledged situation theoretic entities whose instances are **objects** that illustrate the type (e.g., the object type of being a happily married man $[x \mid \exists y \langle \text{married-to}, x, y, + \rangle \wedge \langle \text{man}, x, + \rangle]$, that is, the type of object that is both married to someone and is a man). Similarly, situation types are full-fledged situation theoretic entities whose instances are particular **situations** that illustrate the type (e.g., the type of situation in which someone is talking $[s \mid \langle \text{talking}, x, + \rangle]$).

Constraints

Models generally carry type-level information. In particular, a process model is a structured description of the types of situations in a given process and their temporal (and perhaps spatial or other) interrelationships. That is, a process model depicts the temporal relationships that relevant instances of situation types bear to one another in a given instance of the process. In the context of a process model, the specification of a temporal interrelationship in a given instance of the process is an example of a *constraint*. The precise nature of a constraint varies from one representational framework to another, depending on such things as expressive power and the nature of the information represented. However, a constraint is generally a proposition that indicates something about the essential character of a system **S** — something about the way it *must* behave. More exactly, a constraint indicates a condition k that must hold in **S** under certain other conditions $k_1, \ldots, k_n$. Thus, constraints are inherently type-level information. Situation theory, with its robust type/token distinction, is especially well-suited for the representation of constraint information. In situation theoretic terms, a constraint is a certain kind of relation between types of situations: the type of situation in which the antecedent conditions $k_1, \ldots, k_n$ in question holds and the type of situation in which C consequently holds. Expressing the conditions $k_1, \ldots, k_n$ and k as sets **G**, **G'** of infons, respectively, allows contraints to be expressed explicitly as

$$[s \mid G] \Rightarrow [s' \mid G'].$$

Intuitively, a constraint is said to be *satisfied* in a given system **S** if and only if, whenever there is a situation s of type **T** = $[s \mid G]$, there is a situation s' of type **T'** = $[s' \mid G']$. For instance, the logical constraint that any bachelor is a man is captured situation theoretically as the constraint

$$[s \mid \langle \text{bachelor}, x, + \rangle] \Rightarrow [s \mid \langle \text{man}, x, + \rangle].$$

The notion of a constraint provides the opportunity for a precise definition of a process (i.e., a process type) and, more generally, a *system. A system is a set of constraints designed to achieve some specific end.* Thus, a

process model, in particular, is a representation of a process understood as a system of constraints. The types that constitute the constraints in a process, of course, represent the various activities (at some level of granularity) that constitute the process, and the relations between these activities are the temporal orderings and other salient relations that determine the structure of the process. A process model typically will contain some representational entity E_T to stand for each type T in the process. Semantically, then, T is the *meaning* of the representation E_T. Arrows (as in IDEF3) or some other representational device generally are used to indicate explicitly temporal relations between these types. Richer modeling methods provide means of increasing the expressive capacities of the basic representations of the method, thereby enabling modelers to add more detail to their models. This is the purpose of the IDEF elaboration languages.

The concepts in this section are spelled out in greater detail and are illustrated with an example in Appendix E.

Model Integration[13]

System integration will, for the most part, be a matter of *model* integration because, unless they are highly deterministic and predictable, the activities and processes in a system will have to be represented. The integration of one activity A1 with another A2 means chiefly that A1 is able to respond appropriately to information arising in A2 that is relevant to A1, and vice versa. Thus, A1 must be appropriately connected to potential sources of relevant information in A2 and, unless A1 and A2 are highly deterministic, automated, and predictable, this connection will occur by virtue of there being integrated models of A1 and A2 with which A1 and A2, respectively, are tightly coordinated. Hence, much of this research into system integration depends on the issue of model integration.

Large enterprises today rely on a wide variety of modeling methods and modeling technologies. Typically, models address virtually all aspects of an enterprise, including overall system architecture, product design and life cycle, project management, database design, and so on. The proliferation of modeling methods is primarily a function of their ad hoc origins; many, perhaps most, methods originated in-house to serve specific modeling needs not addressed by existing methods. Another cause of proliferation is the many different kinds of information in a typical enterprise — the relatively static information such as an employee database, dynamic information involved in a model of a manufacturing process, and so on.

Models in general (e.g., CAD models, manufacturing simulation models, physical mock ups, maps, information models, etc.) are effective research and development tools because they enable researchers and analysts

---

[13] Much of this section on model integration was published under the title, "Representation, Information Flow, and Model Integration," in *Enterprise Model Integration* (Petrie, 1992).

to distill just those features essential to their purposes. The researchers and analysts can then conduct their studies unencumbered by the extraneous "noise" of a real-world setting. A good modeling *method* codifies general procedures that embody good modeling practice and provides tools for their construction. An *information modeling method* supplies procedures and tools for modeling a specific kind of information in a given domain — employee information, product design information, manufacturing information, and the like by using certain types of entities and patterns of organization while excluding others. The general pattern of information targeted by a method is its *information type*. (One method, for example, might focus on process information, another on class hierarchies in an organization, and so on.) To express information of that type, a modeling method supplies a *representational medium* — usually some kind of graphical language — *tailored* specifically to that type. In other words, the method establishes semantic connections between the basic elements of the medium and the basic elements of its information type. These connections form the basis of more elaborate semantic connections between complex representations and correspondingly complex pieces of information. Models, in particular, are complex representations; hence, they can be said to *express* or *display* information of the appropriate type by virtue of the semantic connections established by the method.[14]

These general observations suggest in very broad (and rather vague) terms the steps that will likely have to be involved in any effective approach to model integration. These steps are as follow.

1.   Clarify the general structure of the information type targeted by each modeling method used in an enterprise (e.g., isolate the basic kinds of elements involved in that type of information and characterize how those elements combine to generate information).[15]

---

[14] In this paper, models are assumed to be (type-level) *syntactic* entities whose identities are determined by their syntactic structure. The term "model" often picks out a related, semantic concept, namely the *abstraction* that a syntactic model expresses. In this sense of the word, nonisomorphic diagrams and even diagrams in different modeling languages can express the same model (e.g., compare this concept with the idea of the *intended* interpretation of a (syntactic) model below).

[15] This is *not* yet the process of building an ontology that might serve as an *instance* of a given information type. Rather, it is to isolate those general *ontological categories* of the information that the method targets (e.g., objects, times, events, relations, etc.).

2. Draw explicit structural connections between the various information types targeted in the enterprise (e.g., identify or otherwise relate basic elements found in different information types; *objects*, for instance, might be among the constituents of *processes*).

3. Isolate domain-specific connections between, and constraints on, the various things referenced by actual models in the enterprise (e.g., identify information-bearing links between things of one kind and those of another).

4. Guided by those structural and domain specific connections, develop methods for intermodel consistency checking and information propagation and, in particular, methods for determining how the information carried by a model generated by one method is to be transferred to and — as far as possible — expressed in a model generated by another.

Exploring the Problem Space

A model integration problem space consists of three statistically independent dimensions:

1. syntax vs. semantics,

2. global vs. pairwise, and

3. wrappers vs. translators.

There is some latitude in the meanings of these terms; but the discussion above provides a framework for addressing the cluster of issues that this representation of the problem space raises.

Frames, Valuations, and Interpretations

In considering the first dimension, syntax versus semantics, it is useful to put a bit more flesh on the bare bones of Steps 1 and 2. In doing so we risk putting our own possibly controversial spin on the issues; however, the added detail will provide necessary focus for an extended discussion. We do not claim that the proper analyses of "clarify" and "characterize" in Step 1 is "formalize mathematically." The knowledge-representation community acknowledges that mathematical formalization will play a central role in any approach to full-blown enterprise integration and model integration. Accordingly, we suggest that to clarify the general structure of the information type targeted by a modeling method (as in Step 1) is essentially to define what mathematical logicians call a *model*

113

structure or *frame*.[16] In a nutshell, a frame definition specifies an abstract mathematical structure or, in extensional terms, a class of structurally similar mathematical objects. Such mathematical structures are identified with information types. Thus, a frame definition distills the general structure of the information for a given modeling method. Generally speaking, in the case of a method in use, this structure is abstracted from documentation and the study of models generated by the method.

Examples of frames are ubiquitous. A simple frame definition for a modeling method that uses (or can be translated into) a first-order language, for example, might specify simply a pair $\langle D,R \rangle$, where D is a set and R = {R_1, R_2, . . .} is a partition such that each R_n is a subset of $Pow(D^n)$ (i.e., a set of sets of n-tuples of members of D). Membership in some element r of R_1 in such a frame can thus represent simple nonmodal, extensional information to the effect that a certain object has a certain property —one that r might represent. If the information modeled in a domain is subtler and involves modality or some other brand of intentionality, the above simple frame definition could be expanded (using familiar techniques) so that it specifies instead a 5-tuple $\langle D,W,@,dom,P \rangle$, where D and W are disjoint sets, @ a member of W, dom a function from W to $Pow(D)$, and P = {P_1, P_2, ...} is such that each P_n is a set of functions from W to $Pow(D^n)$. This is a more or less typical "possible worlds" frame. Intuitively, D represents the set of all possible objects (or all possible objects in a given enterprise), W represents a set of possible worlds (or possible states of the enterprise), @ is the "actual" world (or current state of the enterprise), dom supplies the objects that exist in each world, and P represents properties- and relations-in-intension as functions from worlds to sets of objects or n-tuples at those worlds. Such frames have been used to characterize, for example, relational database information. The apparatus of worlds and intentional properties and relations permits one to represent possible ways to populate the database (or perhaps, by ordering the worlds, the way it is in fact populated over time): the value of a relation at a world is simply a set of tuples, the instances of that relation in that world (or at that time).

There is no end to the complexity one can build into a frame. More complex frames might introduce objects that capture the structure of large, composite physical systems, or the structure of events and other dynamic objects. In general, any possible mathematical structure is a legitimate choice for representing the structure of information in a given domain, as viewed from the perspective of a given modeling method. Step 1, in brief, is to specify a frame definition for each modeling method used in an enterprise.

Some additional definitions are useful for elaborating on Steps 2 and 3. As alluded to briefly above, a modeling method, in addition to a frame definition, also provides a *valuation definition* that characterizes the

---

[16] Though the former term is more common, the latter is more appropriate in the present context because information models proper are syntactic entities. Therefore, , the term "frame" is used throughout this paper to express this concept.

permissible ways of interpreting the method's representational medium — its models, in particular — in a given frame (i.e., in a given instance of the frame definition for the method). Call any frame that can be assigned to a model **m** by some permissible valuation an *interpretation* of **m**, and given an interpretation I for m, say that m's *domain* relative to I is the (unstructured) class of things (in the broadest sense) in that interpretation.[17] Suppose that the *intended interpretation* of **m** is the interpretation it was designed to represent in a given setting, and its *intended domain* is its domain relative to its intended interpretation. Therefore, for example, if we've designed a database using ER, any frame answering correctly to its boxes, diamonds, and lines is an interpretation of our model. However, its intended interpretation (assuming a relatively noncontroversial frame definition) consists of the actual structured class of object types, relations, and attributes that we have isolated and organized by means of the model, and its intended domain is the unstructured class consisting simply of those types, relations, and attributes. (The unstructured class, for example, does not reveal which attributes are associated with a given object type.)

Step 2 builds on Step 1 by drawing structural connections between information types. For example, upon developing detailed frame definitions for the IDEFØ activity modeling method and the Data Flow Diagram (DFD) method, it may be that IDEFØ activities and DFD data transforms have essentially the same structure. In the same way, it is clear from existing frame definitions that IDEF1 entity classes and ER entity sets (i.e., the things *denoted* by IDEF1 and ER boxes, not the boxes themselves!) are roughly the same kinds of things.

The obvious purpose of Step 2 is to establish formal connections between model types that warrant the translation of the information expressed by a construct in one model into a "synonymous" construct in another. Such connections will then entail actual connections between the elements of the intended interpretations of models of those types. However, as discussed in some detail below, the amount of integration that can be actualized through purely formal connections like this is limited. Rather, even when the formal semantics of a given construct c_1 in one method M_1 matches the semantics of a construct c_2 in another M_2, there is often no guarantee that an occurrence of c_1 in an actual M_1 model m_1 can be translated directly as c_2 in an M_2 model m_2. Rather, there may be *contextual* constraints on the objects in the (intended) interpretations of m_1 and m_2 which restrict translations that might otherwise be formally warranted. (IDEF1 entity classes, as noted, correspond closely to ER entity sets, but because of restrictions on the use of many-to-many relations in IDEF1, certain contexts may require the construction of entity classes that would actually correspond to ER relations.) Conversely, because of specific, context-dependent connections that may exist between elements of those interpretations, the lack of any formal overlap in the semantics of c_1 and c_2 by no means rules out the possibility of there being some kind of systematic information bearing connection between them. Isolating such constraints and connections is the goal of Step 3.

---

[17] We thus want to distinguish clearly between *method semantics*, which consists of a frame definition and a valuation definition, and *model semantics*, which consists in the assignment of a frame to a model by a valuation.

These points are elaborated below. The point here is that the systematic relations between the actual objects in the domains of two or more models uncovered in Steps 2 and 3 can be exploited in Step 4 to establish actual integration mechanisms — in particular, translation mechanisms — between them.

First Dimension

With this background, the integration problem space can be approached directly. The location of an approach to integration in the first dimension appears to mark the extent to which it focuses on syntax as opposed to semantics. However, if syntax has to do with the characterization of representational media (this seems clear) and semantics with the characterization of information types, their interconnections and their relations to those media, then, insofar as the four steps above correctly characterize the general integration process, any approach to integration must involve both syntax and semantics in equal measure. The actual integration procedures developed in Step 4 that yield changes in models (i.e., syntax) will be driven entirely by the intra-method and cross-method semantical connections uncovered in Steps 1 through 3. This is not meant to be profound; it seems to be nothing more than an integration theoretic variation on the truism that translation between languages can proceed only if the meanings of the expressions in those languages, and contexts in which they are used, have been fixed.

Second Dimension

That changes in models will be driven by intra-method and cross-method semantical connections suggests that a global vs. pairwise, or *local*, integration stance should be taken with regard to the second dimension in the problem space.. The global vector in this dimension represents the extent to which an approach provides some *general* framework for integrating models, an approach that is more or less independent of the modeling methods whose models have already been integrated. The more global an approach, the greater its capacity to integrate models generated by other methods. The local vector, by contrast, indicates the extent to which models are merely linked *ad hoc* in pairs as the need arises.

The general integration process sketched above perhaps does not completely determine where an approach to integration should be located in this dimension. However, it certainly suggests that the more global an approach the better. The argument is as follows. Suppose we want to integrate two models m_1 and m_2 from scratch in a local fashion. We begin, according to Step 1, by clarifying each model's information type (i.e., the information type of each model's method), calling these T_1 and T_2, respectively. Step 2 in the process calls for drawing structural connections between T_1 and T_2. In Step 3, domain-specific connections are drawn between elements in the (intended) interpretations I_1 and I_2 of M_1 and M_2. In essence, this requires the construction of a common, overarching *metatheory* that subsumes the two corresponding methods M_1 and M_2 from which m_1 and m_2 are generated in at least the following sense. The metatheory must be capable of expressing:

1.   the syntax of each method,

2.   the information type of each method,

3.   the semantical connections between models and interpretations (especially intended interpretations), and

4.   the structural- and domain-specific connections between models formulated in each method.

Thus, a representational medium *cum* semantics is needed that is at least, roughly, the "union" of M_1 and M_2 in regard to expressive power.

An extreme local approach would be to build such overarching frameworks pairwise as needed. However, there is an obvious problem with a pairwise approach: information relevant to a given model may only arise jointly from several models and may not be contained explicitly in any single model. The idea is just an expression of a simple theorem of propositional logic. Let P be a piece of information expressed by m_1 but not by m_2, and Q a piece of information expressed by m_2 but not by m_1. Furthermore, suppose that neither proposition entails the other. Then, under these conditions, there may be a piece of information R relevant to and expressible in a model m_3 such that $(P \& Q) \dots R$, but such that neither $P \dots R$ nor $Q \dots R$, (e.g., let P be $R \vee S$ and Q be $R \vee {\sim}S$). Hence, to transfer information entailed by several models into a given model that could usefully incorporate that information, there must be an overarching metatheory that subsumes all the models in question. Consequently, a pairwise approach is unsuitable, in general, for model integration within complex systems.

A viable local approach must build up a common metatheory as new methods are integrated with existing integrated methods. However, a problem still remains. While not pairwise, the local approach is still piecemeal and *ad hoc*. Thus, there is no guarantee of extensibility, no guarantee that one's evolving framework for integration can gracefully — if at all — incorporate a new method. Ideally, the framework should have guaranteed extensibility; it should, that is, be a *Global Representational Medium* (*GRM*): a language, together with a well-defined semantics, that is capable of expressing any information in any possible information model. A GRM, therefore, provides an overarching representational framework with the resources for expressing any new information types.

The idea of a GRM might seem naive. We might reasonably hope to develop a representational framework that captures the types of information targeted by *existing* methods, but how can we claim to have a framework capable of expressing the content of any *possible* future methods? The only reasonable grounds for such a claim is that one's representational framework is based upon a general *theory* of information and information flow in complex systems. Just as a good physical theory is able to explain newly discovered physical phenomena under its theoretical structures, so should a robust theory of information have the power to subsume newly discovered types of informational phenomena under *its* theoretical structures. A representational framework based on such a theory will then have the desired characteristics. Our choice of situation theory is based on these considerations.

117

Third Dimension

The description of the third dimension — wrappers vs. translators — is a little harder to clarify. A wrapper approach identifies a communications protocol between models: some set of primitives necessary and sufficient for the desired communication. Each model is responsible for its own envelope that converts its transmissions into this common interchange language and for converting incoming messages from this language. A translator approach, on the other hand, identifies "mapping grammars" between models. In the global approach, the translator provides a general interlingua for all possible modeling languages (e.g., the KIF or the mapping grammars of ANSI Information Resources Dictionary System [IRDS] between specific languages and a normative language based on conceptual graphs[18]). A pairwise approach is to provide mapping grammars directly between pairs of modeling languages as needed (Carnot, 1991).

From these descriptions, it is difficult to tell the difference between the two approaches. In particular, the idea of a "common interchange language" is difficult to distinguish from that of a "general interlingua" for facilitating translation. The real distinction is between *centralized vs. decentralized*, or *mediated vs. unmediated*, approaches to integration. More specifically, location in this dimension marks the extent to which the actual mechanism of integration among sets of models involves translation through a central linguistic hub. In a centralized approach, information is moved between two models m_1 and m_2 through a linguistic intermediary, or *interlingua*, L via two general translation schemes: one between the representational forms of the language of m_1 and L, and another between the representational forms of the language of m_2 and L. L thus acts as an informational clearinghouse, a central hub in the flow of information between m_1 and m_2[19] (at least, as far as possible; one cannot assume that everything expressible in one model is expressible in another). The more global an approach, the more models will be connected by a single hub.

We were able to make some normative claims about the location of an approach to integration in the first two dimensions. Can we do the same here? It is not obvious that we can. If the argument in the preceding section is correct, an effective *design* of the actual integration mechanisms to be implemented in a given system will have to be carried out in a single, overarching representational medium. However, intuitively anyway, the actual mechanisms themselves may or may not make use of an interlingua. It may, in certain circumstances, be more

---

[18] For a brief overview of KIF, see Neches et al., 1991). For the latest incarnation of IRDS that is developing out of the original ISO IRDS documents, see Burkhart et al., 1991].

[19] Compare Neches et al. (1991, p. 39) "To map a knowledge base from one representation language into another, a system builder would use one translator to map the knowledge base into the interchange format and another map from the interchange format back out to the second language."

efficient to transfer information by means of direct translation of the representational forms of a given model into those of another (e.g., by means of a compiler of some sort). The parsing tables for such a compiler will of course be informed at the design stage by what one knows about the two models and their methods as expressed in, ideally, a GRM. However, a representation of that knowledge may not itself be implemented directly.

Integration, Translation, and the Flow of Information

The role of translation in integration raises some particularly important issues. For definiteness, we define a representation r_1 in a model m_1 to be a *translation* of a representation r_2 in a model m_2 just in case the meaning of r_1 (according to the semantic rules of m_1's method) in the intended interpretation of m_1 is the meaning of r_2 (according to the semantic rules of m_2's method) in the intended interpretation of m_2. (For a more formal rendering of this idea in classical model theoretic terms, see Appendix F.)

Now, in some discussions of model integration, it often seems that translation is taken to be very nearly synonymous with model integration; at the least, it often seems to be conceived as the central means by which information is passed from model to model. This is acceptable as long as translation is understood broadly enough. However, as the previous paragraph demonstrates, understanding is subject to constraints. Furthermore, as is commonly the case, the role of translation in integration has some important limitations. The most significant of these for present purposes is that, in general, in complex, heterogeneous systems, there will be much more information carried by a model (or more generally, set of models) than that which is overtly displayed by the particular representations in that model. Hence, a model will generally carry more information than the information that is subject to translation in the above sense.

The most obvious manifestation of this is the case where models of related domains have different expressive capacities or, at least, different intended interpretations. Indeed, this situation will be the norm because different methods typically target different information types. For a very simple example, imagine two modeling languages L_1 and L_2 with the expressive capacities of a monadic first-order language without quantifiers or variables (i.e., essentially, a language with only names and simple predicates and the usual Boolean operators). Furthermore, suppose that L_1's intended domain consists of ordinary individuals in a given system, whereas L_2's consists of the types to which those individuals belong. For simplicity, suppose the predicates of L_1 are the type-denoting names of L_2. Now, consider two models, m_1 and m_2, in these languages that are directed toward the same overall application domain. Although there will be little if any direct overlap in the information that is overtly displayed by the representations in these two models, each may carry much information that is relevant to the other.

For instance, if the statement '*Pa & ~Qa*' occurs in m_1, then, while not precisely translatable into L_2,[20] it nonetheless carries the information that $P \,\grave{A}\, Q$; that is, P is not a subtype of Q (since something that is of type P is not of type Q, namely, *a*).

This example illustrates that the presence of a structural constraint cuts across both models but is expressible in neither. The notion of a constraint varies widely across the modeling, database, and AI communities. One that is particularly well suited to the present context is found in situation theory. According to this theory, information is carried by situations which are themselves (generally) highly structured objects, such as the situation in the U.S. Congress between 9:00 a.m. and 10:00 a.m. on 1 May 1992. Constraints in this theory are essentially type-level conditionals $A \,\textthorn\, C$ to the effect that if there is a situation of type A, there must be a situation of type C. In these terms, the example above exhibits (roughly) the constraint that for any object types S and T, if something is of type S but not of type T, S is not a subtype of T — a fact that presumably follows from a *background theory* of type hierarchies. By virtue of this constraint, we say that the statement '*Pa & ~Qa*' in m_1 *carries the information that* $P \,\grave{A}\, Q$, even though that information is not the *translation* of '*Pa & ~Qa*'. For the two models m_1 and m_2 genuinely to be integrated, we must think in terms beyond simple translation; we must think more generally in terms of the mechanisms of *information flow*.

Once one sees the general pattern embodied in the previous example, it becomes ubiquitous. Constraints are the general conduits through which information flows, the links by which one situation can carry information about another. How things stand in a given model is simply one type of information-bearing situation. The smooth flow of information between such situations (i.e., smooth integration) depends on effective isolation and implementation of the relevant intermodel constraints.

One such set of constraints comprises the metatheoretic constraints embodied in the syntactic and semantic rules of a given modeling method which determine how the representations in a model display information about the situation being modeled. These rules in turn generate further, higher-level constraints that determine whether and, if so, how the information displayed in one model can be displayed in another. Thus, translation is simply a special case of information flow through constraints. For instance, the syntax and semantics of IDEF1 give rise to such constraints as that, in a given modeling situation, a situation of the type **rectangles x and y are connected by arrow Z** involves the type **entity class C_x bears the relation R_Z to C_y**, where C_x and C_y are the entity classes semantically associated with the "variable," or *indeterminate*, IDEF1 rectangles x and y, and R_Z is the relation

---

[20] Except relative to a very coarse-grained semantics, e.g., a semantics that identifies meanings with truth values or even functions from possible worlds to truth values as in classical possible world semantics. We are assuming a much more fine-grained semantics, which we believe is essential to any feasible solution to the problem of model integration.

associated with the indeterminate arrow **Z**. Thus, a modeling situation in which actual IDEF1 rectangles **a** and **b** are connected by a certain arrow **A** means that, in the situation being modeled, C_a bears the relation R_A to C_b by virtue of this constraint. A similar constraint between ER diagrams and entity sets, together with relevant metamodeling constraints, ensures the information carried by IDEF1 constructs of the sort indicated in the constraint carry the same information as certain corresponding constructs in ER. Such metamodeling constraints, coded into a GRM, will form the basis of the translation rules and heuristics that can be used in the process of model integration.

Reasoning systems are a further instance of the same general picture. Inference is the process of unpacking information that is (in general) not explicit in a given representation in accordance with a certain set of logical constraints. Different systems of reasoning (e.g., classical, intuitionistic, non-monatomic, etc.) correspond to different sets of constraints. Thus, a model **m** being updated by an underlying reasoner can be characterized as follows: **m** contains a certain representation **r** (or set of representations) that expresses a piece of information **i** by virtue of the semantic constraints of **m**'s method. A logical constraint embodied in the reasoner dictates that situations involving **i** must involve a further piece of information **i'**. (For example, in classical systems, situations in which *a* is *F* involve the information that *something* is *F*.) It is by virtue of this constraint that we say **r**, and hence **m**, carry the information **i'**, even though neither **r** nor any other representation in **m** explicitly displays **i'**. By adding such a representation, the reasoner updates **m** to a new model **m'** that displays **i'**.

Further examples abound. Most models, especially models of physical systems, assume a large amount of *common-sense* knowledge; thus, they carry far more information than that which is explicitly displayed. Such knowledge can also be thought of in terms of relevant background constraints that enable the flow of information. The same is true of domain-specific background knowledge peculiar to, for example, semiconductor manufacturing or more contextual knowledge of the sort that arises within a particular business, engineering, or manufacturing environment. A budget limitation in the fiscal department, for example, might give rise to constraints on a proposed semiconductor design. Due to one or more of those constraints, a given design model together with background financial information might together carry the information that the design in question is unacceptable.

All these different elements of an integrated class of models fit the same pattern. In each case, systems of constraints determine the information carried by a model (or in general, a set of models) beyond what is explicitly displayed by that model at a given time. These constraints then determine the flow of information from one model to another.

By supplementing the traditional tools of mathematical logic with situation theory, one can gain a more precise grasp of these notions. In particular, the theory provides us a notion of a *unit* of information **i** (and of more complex pieces of information built up therefrom) that correspond naturally to ordinary language descriptions of system information and information flow. Toward providing a more rigorous framework for the formal analysis of integration, we propose the following definitions (space limitations prohibit more detailed renderings). Suppose a

121

model **m** *displays*, or *expresses*, the information **i** (at a time, or over an interval, *t*) just in case there is a representation **r** in **m** and a sequence of semantic constraints $C_1, ..., C_n$ from **m**'s method such that **i** is the (intended) interpretation of **r** according to $C_1, ..., C_n$. A set S of models $\{m_1, ..., m_n\}$ *conveys*, or *carries*, the information **i** (at, or over, *t*), just in case there are pieces of information $i_1, .. i_n$ and constraints $C_1, .. , C_m$ such that for all **j** such that $1 \pounds j \pounds n$, m_j displays the information i_j, and $i_1, ... i_n$ together with $C_1, .. , C_m$ entail **i**.[21] (Consequently, we might borrow notation for the related concept in mathematical model theory and write this as $\{i_1, ... i_n, C_1, ..., C_n\} \models$ **i**.) Let S be a set of models, and let **m** be a model that does not carry a certain piece of information **i**. Then we can say that the information **i** *flows* from S to **m** over a given interval *t* just in case S carries **i** over some initial segment of *t*, and some agent or mechanism (e.g., a reasoner, a constraint propagator, etc.) that is correctly attuned to the relevant constraints (during *t*) updates **m** to a model **m¢** that displays **i**.

Using these ideas, we can define a notion of an integrated system of models. We take as primitive the idea of a piece of information being *relevant* to a model over an interval *t*. A system **M** of models is *fully integrated* for any subset S Õ **M** and for any m Œ **M**, if S carries the information **i** at *t*, and **i** is relevant to **m** at *t*, then **i** flows from S to **m** in a reasonable time (relative to the needs of the system). Finally, by making certain assumptions, we can define a notion of the *degree* of integration that exists in a system of models. Specifically, suppose there is only a finite amount of information in a given domain that is relevant to a model at a given time. Then, we can define the degree of integration in a system **M** of models over *t* to be the ratio of the number of pieces of relevant information that actually flow in **M** over *t* to the number of pieces of information *per se* that are relevant to the models in **M** over *t*.

Ontology-Driven Information Integration

This section describes in detail the general vision of ontology-driven information integration being developed in this project. In particular, the development of ontology libraries, their use in enterprise modeling, and their integration is discussed.

Three Roadblocks to Enterprise Information Integration

Three roadblocks stand in the way of enterprise integration: semantic inaccessibility, logical disconnectedness, and consistency maintenance. Each of these roadblocks are discussed in detail in the following paragraphs.

---

[21] Compare the parallel distinction in Devlin (1991) between a situation carrying information in virtue of being of a certain type (this corresponds to a model displaying a piece of information) and it carrying information in virtue of a constraint.

Semantic Inaccessibility. As noted above, the challenge of integration begins with the fact that each application in an enterprise tends to keep its own private repository of information. The problem with this is not necessarily that the *data* — that is, the various forms of representation used in some aspect of an enterprise to carry information — generated by these tools are inaccessible. Presently tools and techniques are available which can transfer and convert raw data files from one format or platform to another. rather, the problem is that a complex representation like a data model or business process model carries the information it does via an established, systematic connection between the components of the representation and the world. It is this connection that determines the *semantic content* of the representation.

Typically, however, the semantic rules of a representation system for a given application and the semantic intentions of the application designers are not advertised or in any way accessible to other agents in the organization, thus making it difficult, even impossible, for such agents to determine the semantic content of a database. This problem, called the *problem of semantic inaccessibility* manifests itself superficially in the forms of unresolved ambiguity (as when the same term is used in different contexts with different meanings) and unidentified redundancy (as when different terms are used in different contexts with the same meanings). But these are just symptoms; the real problem is how to *determine* the presence of ambiguity, redundancy, and their ilk in the first place. That is, more generally, how is it possible to access the semantics of enterprise data across different contexts? How is it possible to fix their semantics objectively in a way that permits accurate interpretation by agents outside the immediate context of those data? Without this capacity, the kind of coordination between enterprise teams necessary for a truly integrated environment is not possible.

Logical Disconnectedness. Even given a solution to the problem of semantic interpretability, a further problem impedes full cooperation among disparate teams. Suppose, for instance, we have determined that a certain representation **R1** in a design model **M1** is semantically equivalent to a representation **R2** in a given analysis model **M2**, that both **R1** and **R2** stand for the same entity— a proposed shuttle part **P**, as in the example above. Thus, the models **M1** and **M2** both carry information about **P**. Suppose now that the information about **P** in **M2** is updated as in our example. This requires a change in the information carried about **P** in **M1**. The fact that it is known that **R1** and **R2** are semantically equivalent in and of itself has no bearing whatever on whether the implications of the change in **M2** will be propagated to . More generally, then, the problem is that the constraints between the particular pieces of information generated by various tools both within and across enterprise contexts are rarely maintained; this has been termed the *problem of logical disconnectedness*.

Consistency Maintenance. Full integration, however, requires more than the maintenance of constraints across enterprise domains because there is no guarantee that, for example, a model **M1** will remain internally consistent when it is updated in light of a change in another **M2**. More generally, a dynamic body of enterprise information is always subject to inconsistency. The simplest way in which this can occur is for a database to be updated with information that conflicts with information already in the database. The problem is exacerbated in a

distributed environment because inconsistency may not arise simply within a single database but rather across two or more distinct databases; two databases **B1** and **B2** may both be consistent internally, yet inconsistent with one another. This fact raises the question of how the source of inconsistency is to be detected and, once detected, managed within an integrated environment. The problem of inconsistency management is particularly important because inconsistency pollutes all the information in a database; anything at all can be deduced from an inconsistent database (in systems founded on classical logic, anyway), and nothing that follows from such a database is reliable. Hence, mechanisms must be in place for isolating the sources of inconsistency in a set of related databases and returning the set to a stable, consistent state. The detection and management of inconsistency has been titled the *problem of consistency maintenance.*

Ontology and Integration

As noted, the proposed approach is built around the notion of ontology libraries. This section indicates how the ontologies can be used to overcome the three roadblocks to integration.

Ontology Libraries. Concisely stated, an ontology is an inventory of the kinds of things that are presumed to exist in a given domain (e.g., mission planning, space shuttle design, etc.) together with a description of the salient properties of those things and the salient relations that hold among them. Thus, an ontology provides a *representational vocabulary* (Gruber, 1992) for a given domain together with a set of definitions, or *axioms*, that sufficiently constrain the meanings of the terms in that vocabulary to enable consistent interpretation of data using that vocabulary.

One of the original motivations for the development of ontologies was born out of an analogy with software engineering. Often in the development of software the same kinds of routines (e.g., in the design of user interfaces) are used again and again in different programs by (in general) different programmers. In earlier times, enormous amounts of time and effort were lost for lack of the ability to reuse previous work. Recognition of this problem has led over time to the development of extensive *libraries* that contain often-used routines which programmers can simply call into their programs rather than having to duplicate existing code. The use of knowledge-based applications in engineering and manufacturing systems faces the same sort of problem. Consider for instance a knowledge-based manufacturing process planning application. An effective manufacturing process plan in general requires extensive knowledge of, for example, relevant numerical computing (NC) machines, part geometry, and general knowledge about how these specific pieces of information can be conjoined into an efficient process plan. The process of collecting such knowledge is often lengthy, expensive, and, once coded into an application, difficult to reuse. However, much of this knowledge is general, hence *reusable.* Building off the analogy with programming libraries, significant effort is now being expended to construct reusable repositories of such knowledge in the form of ontology libraries (i.e., large, reusable, customizable knowledge bases of structured, domain-specific ontological information). By making such libraries generally available to developers, much of the

arduous process of collecting domain knowledge can be avoided, and the costs of development of custom knowledge-based applications can be significantly diminished.

A serious problem facing the idea of reusable ontologies is the proliferation of representation languages. If a given ontology library is developed in a particular representation language $L1$ (e.g., conceptual graphs,), it is usable by an application that uses a different language $L2$ (e.g., LOOM) only after translation into $L2$, a task which, though not as arduous as building the ontology from scratch, nonetheless would require a significant amount of effort. Furthermore, even if one were to accomplish such a translation, without some sort of objective principles of translation from $L1$ to $L2$, there would still remain questions about its consistency vis-a-vis other translation methods.

Fortunately, there is an alternative to manual translation. The Shared Reusable Knowledge Bases project in the Knowledge Sharing Effort (KSE) is developing a mechanism known as *Ontolingua* for translating ontologies between representation languages (Gruber, 1992). Currently, translators exist for a number of different representation languages. Writing a translator, of course, is no simple matter; however, once written, the task is complete, unlike the task of translating new ontologies. Furthermore, once a translator is built it can be published and tested objectively for semantic accuracy, debugged, and distributed widely, thus ensuring semantic consistency of all translations from one representation language to another.

Ontology, Constraints, and Information Integration. Reuse, however, is only the first, most basic application of ontologies. Ontologies provide a major part of the solution to the problem of information integration. Consider first the problem of semantic inaccessibility. This problem arises in two forms: when an agent A_1 — for example, a certain application— in a given domain D_1 maintains a database B_1 that carries a piece of information I needed by a human or computer agent A_2 in a different domain. The first form of the problem of semantic accessibility is that the information in B_1 needed by A_2 might be represented by A_1 in a form that cannot be interpreted by A_2. This problem can be effectively addressed by means of the translation technology just discussed. The second, more serious problem is that the information in question may not be fully accessible to A_2 because A_2 lacks the *background knowledge* needed to extract it from B_1, regardless of the represented form of the information in B_1. Background knowledge, or *domain knowledge*, is knowledge that is not explicitly represented in the database being accessed but is either hard-coded into domain applications that access the database directly or sufficiently understood by domain experts using the application, making explicit or hard-coded representation unnecessary. An ontology for the domain D_1 of B_1 supplies the needed background knowledge (i.e., a representational vocabulary for D_1 and relevant axioms) to obtain the requisite information.

How, exactly, is this accomplished? The solution we propose is simple but requires a fair amount of ground clearing. Information is extracted by means of constraints. Constraints have been described in broad and somewhat vague terms as dependencies of some sort (e.g., functional, existential, causal, etc.). More precisely, a

constraint is simply any statement that *must* hold in a system; the system is constrained to behave in accordance with that statement. Typically, however, constraints indicate some sort of distinctive logical connection (in the broad sense of "logic") that must be maintained between enterprise objects, attributes, activities, and so forth. For example, it is a temporal constraint on the concept *before* that if activity T_1 occurs before activity T_2, and T_2 occurs before T_3, then T_1 also occurs before T_3. A functional constraint indicates a functional relation between (typically) two attributes, for example, the temperature and pressure of an enclosed gas. An existential constraint indicates a relation of existential dependence between two particular objects or two *kinds* of objects (i.e., in the first case, that a particular object O_1 (a car, say) cannot exist if another O_2 (its chassis) does not, or in the second case, that *every instance* of a given kind of object K_1 (the kind *car*) requires the existence of an *instance* of another kind of object K_2 (the kind *chassis*).

Notably, constraints are generally *conditional*.[22] That is, they are of the form, "*If* such and such is the case, *then* so and so must be the case as well." This is obvious in the temporal example above. To see that functional constraints are essentially conditional also, note that awareness of a functional relationship between two quantities typically will be significant because it enables one to calculate an unknown value as a function of a known value. Thus, the functional relation between temperature and pressure, expressed as $Temp(g) = f(Pr(g))$, indicates that the temperature of a given volume of gas can be deduced from its pressure. Thus, the constraint in question is best represented in the form of a conditional whose antecedent explicitly displays the known value (as given by some measuring device, say), and whose consequent displays the information that can be extracted from that knowledge, that is, the conditional "If $Pr(g) = n$, then $Temp(g) = f(n)$," (see Appendix E),

$$(\Rightarrow (= (\text{pressure } g) \text{ n}) (= (\text{temp } g) (\text{f } n))).$$

Finally, the existential dependencies between O_1 and O_2, and K_1 and K_2, respectively, are also clearly conditional: "If $Exists(O_1)$, then $Exists(O_2)$," and "If $K_1(x)$ then there exists a y such that $K_2(y)$," or more formally again,

$$(\Rightarrow (\textbf{exists } x (= x \ O_1)) (\textbf{exists } y (= y \ O_2)))$$

$$(\Rightarrow (K_1 \ x) (\textbf{exists } y (K_2 \ y))).$$

---

[22] This is not entirely true because there may be global constraints that apply unconditionally to the entire system (though, of course, a global constraint C can be treated trivially as a conditional by taking it to be of the form "If TRUE then C"). Such constraints are perhaps best enforced locally in each relevant application. In any case, the treatment of global constraints is an issue that has to be handled with some delicacy. Investigation of the issues will be taken up in Phase II of this project.

Constraints, by virtue of their conditional form, act as "conduits" through which information flows. Generally speaking, if a (human or computer) agent A^* is "attuned"[23] to a constraint ($\Rightarrow$ j y) (call this constraint "C"), knowledge that j holds enables A^* to extract the information that y holds as well. Suppose that the information j holds only arises in a certain domain D, and that the information y holds is relevant to some other domain D'. Then C is what we can call an *interdomain constraint*. Suppose agent A^* is attuned to C and knows also of its relevance to applications using information about D'. Suppose also that A^* comes to know from an agent A_1 in domain D that j holds. Then, if A^* proceeds to pass the information y holds to an agent A_2 in D', information is said to *flow* from A_1 to A_2. C is the *conduit* for that information, and A^* is the *mechanism* for the flow.

We can now begin to answer the question raised above about how ontologies enable the extraction of information by the agent A_2 from the database B_1. The background information needed to derive the necessary piece of information I comes in the form of a constraint that links some piece of information I' somewhere in the system (typically in A_2's own database) to I. That is, I can be relevant to A_2 only if there is some constraint that makes it so, a constraint to the effect that:

> If I', then I.

This constraint itself is ontological information, albeit an interdomain rather than *intra*domain constraint; hence, it is not likely to be found in an ontology for D_1 proper.[24] However, analysis of such an ontology (customized as necessary for the given context) together with an ontology for A_2's own domain will — if the ontologies are themselves sufficiently complete — reveal the logical connections between enterprise elements that it embodies.

For A_2 to be apprised of the desired information, I only requires that an agent A^*, attuned to the constraint in question, serve as a mechanism that causes the information in question to flow from A_1 to A_2. Such an agent lies at the center of the ontology-driven architecture. The heart of the architecture is an integration manager containing, ideally, all relevant interdomain constraints that connect the enterprise objects — part designs, mission plans, and so on — about which information is being kept in a distributed system. The integration manager is able to communicate via appropriate translators (when necessary) with every other application in the system and is apprised of changes in the information in any given database. Should this information be relevant to some constraint, the integration manager checks to see whether — perhaps together with previously stored information — it triggers any

---

[23] We use the language of situation theory here. See especially Barwise and Perry (1983) and Devlin (1991).

[24] Notably, this is a relative distinction; an interdomain constraint can become an intradomain constraint from a higher perspective.

registered constraints. If so, the constraint is fired and the new information that is extracted is propagated to all relevant applications. (Of course, it is the responsibility of the application to deal with the information appropriately once apprised of it.) If no constraint is fired, the information is stored in the integration manager for future reference. Although useful integration can begin almost immediately upon implementation of this architecture, how much information in each application database will have to be replicated in the integration manager for *optimal* integration (i.e., integration that optimally balances usefulness against computational complexity) is an open issue.

This proposed solution to the problem of semantic accessibility provides a solution to the problem of logical disconnectedness. The proposed solution to the first problem is essentially to make other databases accessible by establishing relevant logical connections between them via a centralized integration manager. In one sense, this is a limited solution because it depends upon *previously discovered* constraints in an enterprise. However, the process of constraint discovery is not automatic. Typically, constraints will not be free, any more than ontologies themselves. As with the development of ontologies in general, there will be no way at the outset to determine precisely which constraints are going to be needed to enable significant enterprise integration. Thus, robustness will necessarily grow incrementally as appropriate constraints are added to the integration manager. To accommodate this inevitable limitation, it will be possible for human agents to browse enterprise ontologies directly in their preferred format (via Ontolingua) to enable them to "manually" interpret other databases across enterprise contexts and to derive interdomain constraints from them to be added to the integration manager. This method will be the preferred method of constraint discovery.

Finally, regarding the problem of consistency maintenance, when logical connections between enterprise elements are properly mapped and registered as constraints in the integration manager, consistency across applications will be maintained via local consistency checking within the integration manager itself. For reasons noted previously, the robustness of consistency maintenance will also increase with the number of registered constraints.

Implications for Standards Work

It is important to note the broader implications of model integration for the international standards community. The picture of the KIF kernel and its extensions can be thought of more generally as a picture of a common logical kernel around which is built a large family of knowledge representation (KR) and enterprise modeling (EM) languages, and what we might call *metamodeling* languages. In addition to KIF in its more expressive guises, these languages include the ANSI IRDS, PDES SUMM, British specification language Z ("zed"), EXPRESS, SQL3 (an extension of the Structured Query Language designed to be a general specification language), CASE Data Interchange Format (CDIF), and Object Query Language (OQL). Thus, work done toward clarification of the KIF kernel and its extensions, and how those extensions are related logically, can be leveraged directly in

support of the problem of integrating this array of languages that are, for important reasons, the focus of international standards efforts.

The development of the wide variety of KR and EM languages (including CASE languages and languages for object-oriented programming and design) arose largely out of the need for *specialized* representational media (i.e., languages whose syntax and/or semantics were designed to capture information — typically of a certain specialized sort — in a form that, for the purposes intended for those languages, are particularly useful or evocative). Thus, much of the work on semantic net formalisms has been motivated by a certain model of memory that suggests these languages correspond particularly well to the internal representational structures and reasoning capacities of human agents. Similarly, an EM language is designed to represent only a certain specific *type* of information; hence, keeping a modeler focused on that type of information typically require restrictive syntactic rules and limited expressive power. Thus, the ER and IDEF1X languages are designed specifically to express the general structure of the concepts, attributes, and relations of a database schema; consequently, they have no mechanisms of quantification, modal operators, and the like. Thus, the focus of a modeler using these languages is trained directly upon only those notions and structures appropriate to relational database design. Problems arise only if one attempts to use the languages outside their intended domain to express information for which they were not designed.[25]

The major strengths of these languages and their corresponding tools derive from their high-level application-oriented features. Their weaknesses derive from the fact that other languages have radically different ways of defining similar features. Thus, there is no way to interconnect different high-level tools based on those languages to each other or to programs written in conventional languages while preserving the high-level environment. Rather, when programmers try to interconnect them, they are forced to switch from a very high-level application-oriented code to the lowest level of bit-twiddling conversion routines.

Therefore, a standard framework for the interchange of information is needed across a far larger class of languages — and a broader class of applications — than simply the KR languages and tools KIF typically has to use. Ironically, the realization of the need for such a framework to support the integration of disparate KR and EM languages in and across domains has led to a new sort of proliferation. KIF is just one of several, typically more expressive meta-modeling languages based on logic that have been designed to integrate some cluster of specialized languages or, more generally, to be expressive enough to capture any domain information that might occur in any domain model or system specification. In particular, in addition to KIF, other languages playing similar roles

---

[25] Failure to realize this problem has increased the proliferation of EM languages in a particular unfortunate fashion because it has often led to the development of ad hoc, bastardized extensions of original languages that possess no clear syntactic rules for their use and no clear semantic rules for their interpretation.

include the normative language (based on Sowa's [1984] Conceptual Graphs) of the ANSI IRDS CS, the PDES SUMM, the CDIF, Z , the OQL of the Object Data Management Group, the ISO SQL3 language, and EXPRESS. The potential benefits of such *meta-modeling* languages are significant.[26] By providing a unified representation of the information in a given domain, a meta-model provides an objective basis for interpreting the specialized models and representations within that domain in another, and hence an objective basis for communication between agents and tools in those domains. Furthermore, because a meta-model separates the *contents* of a model or specification from the vagaries of specialized languages, technology, and interfaces, it provides a robust foundation for reuse in both system and software design and specification (Semantic Unification Meta-Model Dictionary/Methodology Committee [SUMM], 1992; International Standards Organization [ISO], 1987; and Burkhart et al., 1991).

However, because of the proliferation of such languages, a unified representation of the information in a given system (or design or specification) may not be available to other individuals and organizations who do not know the language. Hence, the benefits of such unified representations, while still substantial, remain localized to those groups that "speak" the same meta-modeling language. Thus, the situation, in this respect, is directly parallel to the problem of proliferation among specialized KR and EM languages. However, in contrast to the case of specialized languages where proliferation is desirable, meta-language proliferation is to be avoided because the point of a meta-modeling language is to capture the system information and specifications of particular models in a unified, *non*specialized form to support consistency, sharing, and reuse. Thus, the definition of logic-based meta-modeling framework is an ideal target for standardization.

This has not gone unrecognized. The ANSI X3 committee, in particular, has been urging the development of a logic-based approach, not only for the IRDS CS but for object-oriented programming systems as well. Additionally, the ANSI X3T2 committee is profiting from the efforts of the KSE by adopting KIF as the language for data interchange. Furthermore, these ANSI and ARPA efforts are being coordinated with the ISO Special Working Group on Conceptual Schema and Data Modeling Facilities. Finally, another response to the need for logic-based model integration led to the development of the SUMM. The SUMM was developed between 1990 and

---

[26] It is not assumed that the distinction between meta-modeling languages on the one hand and ER and EM languages on the other is in all cases clear and unambiguous. Meta-modeling languages usually *could* be used as knowledge representation or EM languages but would typically be either too austere or unstructured to be useful. Conversely, some languages that are best classified as knowledge representation or EM languages might have the logical expressive power of a meta-modeling language; however, this power is often couched in a specialized syntax that would make it cumbersome and ill-suited to the meta-modeling task.

1992 by the Dictionary Methodology Committee of the IGES/PDES Organization (IPO).[27] Its approach is similar to the IICE NIRS; indeed, Dr. Menzel of KBSI contributed significantly to the development of the SUMM in its final six months of development. The SUMM was developed to provide a logic-based formalism for exchanging data models between the EXPRESS, IDEF-1X, and NIAM modeling languages, which were then in use by STEP. Early versions of the SUMM were presented to various national and international standards agencies, including ANSI X3H4, ANSI X3T2, and ISO/IEC JTC1 SC21. In 1992, SC21 initiated a formal question among its member nations whether work should be performed to make the SUMM an international standard. The result of this question was the initiation of the Special Working Group on Conceptual Schema and Data Modeling Facilities. ANSI X3T2 included the SUMM among the documents to be included in the U.S. contribution to this working group. Meanwhile ISO TC184/SC4/WG5, which is responsible for the development of EXPRESS and methods for its use within STEP, is using the SUMM as a benchmark against which to develop formal meta-models of EXPRESS, both to improve the language and to unify it with other data-modeling languages.

An Approach

This section sketches an approach to the problem of integrating multiple modeling languages. The first step consists in the formalization of existing languages, where necessary. (The IRDS normative language and the SUMM, for example, are already fully formalized.) These formalizations should be based on existing documentation and, where possible, discussions with developers and prominent users. The products of the formalization of a given language would include a clearly defined syntax, including its basic lexicon and its grammatical rules, and a rigorous model theory, including a clear measure of the representational power of the language (e.g., whether it is first-order or higher-order; whether it includes number theory, set theory, etc.).

Upon completion of these formalizations, the next step is to analyze the languages to identify, if possible, a common logical core (CLC) shared by all the languages. This CLC should be expressed in a standard logical language. It should then be determined how this core must be extended for each language (e.g., by the addition of set theory, mereology, fine-grained intentions, etc.), and translational mappings between the logical language and each language must be defined. The results of these formalizations must then be analyzed to provide rigorous comparisons of the representational power of the languages. This task may not be possible directly in terms of the formalizations because a number of the languages in question are nonstandard in form. For this reason (among others), there must be a general formal framework — the NIRS — for describing the way information is stored or transmitted by a particular representation to facilitate the formal comparison of different representation languages.

---

[27] Initial Graphics Exchange Specification (IGES) is an international standard for the exchange of drafting information, and PDES is the U.S. contribution to Standard for the Exchange of Product Model Data (STEP), which is the informal name for ISO 10303, a Draft International Standard being developed by ISO TC184SC4.

These results concerning the representational power of a given language can then be used to determine the extent to which knowledge can be shared between knowledge bases in different modeling languages via their respective mappings into the extension of the logical core that constitutes their "representational overlap." It is expected that this overlap between languages typically will be quite high, given the nature of these languages.

These analyses will result in a family of extensions of the CLC, all expressed in a standard logical notation, which serves as the basis for integration between languages via knowledge-sharing interfaces. The possibility of clearly delineated, knowledge-sharing interfaces between modeling languages is crucial because they are the key to overcoming the problem of multiple modeling languages. They make the integration of the languages possible and, thus, the sharing and reuse of enterprise models, designs, specifications, and the like. expressed in those languages.

System Integration

An Intuitive Account

This section provides an intuitive notion of integration, a notion of a system, and an explanation of what it means for the system to be integrated.

1.   System: A group or set of objects united by some form or regular interaction or interdependence to perform a specified function. A system is characterized by its:

> Agents
>
> Resources
>
> Products
>
> Goals
>
> Structure
>
> Behavior
>
> Constraint Set

2.   Agent: An individual capable of performing some action.

3.   Agent Types: An abstraction of agents based on what they are authorized/ permitted/mandated to do or capable of doing.

4.   Resource: An individual some agent can use. "Usage" can be physical (e.g., machine tools), exploitation of knowledge about the system or environment (e.g., information system), of services of agents who do not belong to the system (e.g., consultants).

5.   Resource Type: Resources abstracted on usage type and/or capacity.

6. Product: An artifact of the system (e.g., design drawings, process flow charts, as well as the final product offered to the customer).

7. Product Type: An abstraction based on the function/use of the artifact.

   The difference between a resource and a product is not in their purpose or use, but their origin. In a sense, every product will be put to some use and, hence, is a resource. (For example, the final product sold to the customer is a resource to attain a higher goal of making profits.) Products are generated within the system (hence, the notion of "artifact"), whereas resources are generated outside the system but are used by the system. This difference creates an entirely new way to distinguish between resources and products.

   There is a strong existential dependency between a product and the system, whereas there is no existential dependency between a resource and the system. Of course, an object will gain the attribute "resource," however, it is used by a system and in that sense is system dependent. That is, the property "resource" is system dependent but the existence of the object itself is not.

8. Goals: Desired state of affairs in the world. Modeled as type of situations that support the desired state of affairs (e.g., type of situations in which profits exist). Situation types are needed because the desired state of affairs can be realized in more than one possible way, leading to different situations.

   A system can have multiple goals, and it is possible that some of these goals have some interaction. For example, a nonprofit organization can have goals: to provide service to the maximum number of customers and to operate within a budget. The first goal can be viewed as an optimization function, the second as a constraint.

   In general, a unified scale of measuring the extent of realization of these goals is needed. For example, an enterprise can have goals: maximize profits and increase market share. These two goals can be contradictory at some point (lower prices implies larger market share but also implies smaller profit). Alternate strategies can be evaluated in terms of maximizing profits so that the market share is not less than x% or maximizing market share so that profits exist.

9. Structure: The situation describing all static information about the system. It will contain information regarding the properties of the agents, resources, products, system, environment, and their interactions. For example, the structure will contain the hierarchy in the enterprise, rules of the enterprise (essentially, IDEF9 constraints), and so forth. A rule in the organization such as "pressure in the pressure vessel should not exceed 200

psi" is part of the structure of the system. The rule, per se, is static information about the enterprise, even though it can potentially influence the behavior of the system.

10. Behavior: Set of all agent roles. The behavior of an agent is formalized as a pair <S, A>, where **A** is the action type that the agent exhibits in situation type **S**. For example, the situation type in which the pressure in the pressure vessel exceeds 200 psi (**S**) involves the situation in which the operator releases the safety valve (**A**).

There are several possible behaviors of a system. For example, in the situation in which the pressure in the pressure vessel exceeds 200 psi, the operator can shut off the compressor instead of releasing the safety valve.

Integration is concerned with exhibiting optimal behavior with respect to the goals of the system. For instance, in the above example, integration necessitates that the operator be aware of the consequences of the pressure in the pressure vessel exceeding 200 psi and exhibit appropriate behavior considering this knowledge and the system's goal not to cause any damage or explosions.

11. Constraint Set: An information flow enabling element. Constraints enable us to deduce the situation type that will result from a given state of affairs. Some constraint types identified by Devlin (1991) are:

a. Nomic Constraints: Constraints that result because of the existing nature of the system (and the world). For example, the situation type in which the pressure in the pressure vessel is greater than 300 psi and continues to increase involves the situation in which the pressure vessel blows.

b. Normative Constraints: Constraints that are a result of norms or conventions. These are not as universally applicable as nomic constraints but are equally important. Examples of normative constraints include constraints on tolerance that the design engineer encodes in a blue print that the production engineer has to decipher to produce the component according to intended specifications. Linguistic constraints are also normative.

An important feature of this constraint set is its open-endedness. That is, there usually will be a potentially infinite number of constraints even in a simple system. The problem is somewhat similar to the frame problem of artificial intelligence. Just as in the frame problem, this open-endedness, although it has interesting theoretical and philosophical ramifications, has little practical implications. It is assumed that only the known constraints are important and necessary (and this constraint set is updated as new constraints are discovered).

Integration

An integrated system is a system in which the system's behavior is optimal with respect to the goals of the system. This definition implies the following stipulations.

1. An agent is aware of all the relevant constraints. Relevant constraints are:

    a. Constraints that are triggered by any action of an agent. This definition basically implies that agents are aware of the consequences of all their actions. For example, the operator should be aware of the consequences of a failure to take action when pressure in the pressure vessel exceeding 200 psi. In this case, inaction on the part of the operator itself is an action.

    b. Constraints that are triggered by the actions of other agents or by any other changes in the system that effect the work environment of the agent.

2. An agent is capable of perceiving and propagating this change through the constraint network and making deductions about the state of the system.

3. An agent is capable of deciding which actions are optimal with respect to the goals of the system.

4. An agent has the ability to translate decisions into actions.

From the above discussion, the following criteria of integration have been developed:

1. Awareness of relevant constraints.

2. Ability to perceive changes in the system and propagate these changes through the constraint network.

3. Ability to decide which of the various actions is optimal with respect to the goals of the system.

4. Ability to translate decisions into actions.

Theoretical Measure of Integration. As explained, one criterion of integration is attunement to the relevant constraints. Based on this criterion, a theoretical measure of integration can be defined as:

$$\text{Measure of Integration} = \frac{\sum_{\text{All agents}} (\text{\# of relevant constraints the agent is aware of})}{\sum_{\text{All agents}} (\text{\# of relevant constraints})}$$

Practical Considerations

1. As already explained, there is open-endedness with respect to the set of constraints. Consequently, even the set of relevant constraints can be infinite. Thus, theoretically, the sum in the denominator tends to infinity and, because the knowledge of any cognitive agent is bounded (Theory of Bounded Rationality), the measure of integration tends to zero and, as such, is not very useful. However, most of these constraints are trivial and not encountered in practice. The first practical accommodation made is restricting the set of constraints to all constraints that were known and important about the system and continuing to update the system with additional constraints as they are experienced or discovered.

2. Another practical problem is how to count the relevant constraints that an agent is aware of. Awareness to relevant constraints can only be measured in terms of the behavior exhibited. Just like the Spring Constant of a spring can be measured by deforming the spring, awareness to relevant constraints can be measured by introducing change (chaos) in the system and observing the behavior of the agent. Therefore, one practical measure of integration in a system could be to introduce a change in the system and look for the number of constraints that are accounted for in the agent's behavior. Based on this criterion, a Practical Measure of Integration (PMI) is defined as:

$$PMI = \frac{\displaystyle\sum_{\substack{All\ changes \\ introduced}} \sum_{All\ agents} (\#\ of\ relevant\ constraints\ accounted\ for\ in\ agent's\ behavior)}{\displaystyle\sum_{\substack{All\ changes \\ introduced}} \sum_{All\ agents} (\#\ of\ relevant\ constraints)},$$

where PMI is the practical measure of integration. The implicit assumption in the above measure is that changes can be planned and artificially introduced into the system. Furthermore, the quality of the measure is dependent on the completeness of the test suites introduced. That is, the changes should be varied enough to cover a significant number of scenarios. Simulation is one possible tool to overcome the limitations of introducing changes into the real system.

Formal Theory

Basic Theory

Let $C = [S_1 \Rightarrow S_2]$. Suppose that (1) A_1 knows there is an $s_1{:}S_1$ in S, (2) A_1 communicates to A there is an $s_1{:}S_1$ in S, and (3) A_2 comes to know there is an $s_2{:}S_2$ in S by virtue of A's attunement to C. Then, the information

that there is an s_2:S_2 is said to *flow* from A_1 to A_2 (via C) over t. C is said to be the *conduit* of the information flow, and A is said to be the *mechanism* of the flow.

A constraint $C = [S_1 \Rightarrow S_2]$ is *implemented* in system S between agents A_1 and A_2 over t just in case, if A_1 were to know there is an s_1:S_1 in S at any point in t, the information that there is an s_2:S_2 would flow from A_1 to A_2 via C over t, or over some subsequent time t' in t.

A constraint C is *implemented* in a system S over t just in case it is implemented between two (possibly identical) agents in S over t.

If $C = [S_1 \Rightarrow S_2]$, C is *triggered* in S just in case there is a situation s_1:S_1 in S.

A constraint $C = [S_1 \Rightarrow S_2]$ is *relevant* to a system S relative to agents A_1 and A_2 over t just in case it is possible that C be implemented in S between A_1 and A_2, and, if it were, its being implemented would significantly affect the ability of S to achieve one or more of its goals over t.

The utility of a set M of constraints relative to S over t (util(M,S,t)) = (the benefits of implementing and maintaining every constraint in M in S over t) - (the cost of implementing and maintaining every constraint in M in S over t).

The utility of a sequence $M = \langle M_1,...,M_n \rangle$ of sets of constraints relative to S over a sequence of contiguous intervals $t = \langle t_1,...t_n \rangle$ (util(M,S,t)) = S_i(util(M_i,S,t_i)).

We will often speak of the utility of a sequence M relative to S over $t \approx \langle t_1, ..., t_n \rangle$, indicating by "$t$" the interval that comprises the intervals in the sequence $\langle t_1, ..., t_n \rangle$, similarly for the other notions below. (So $t \approx \langle t_1, ..., t_n \rangle$ iff t is a sequence of contiguous intervals and $t = \cup\{t_1, ..., t_n\}$.) Where no confusion will result, explicit reference to sequences of intervals is sometimes suppressed and referred only to the subsuming interval t. Note that the point here is that a given interval can be decomposed into any number of different subintervals.

A sequence $M = \langle M_1,...,M_n \rangle$ of sets of constraints is *beneficial* to a system S over $t \approx \langle t_1, ..., t_n \rangle$ iff the utility of M over is positive.

A sequence $M = \langle M_1,...,M_n \rangle$ of sets of constraints is *more beneficial to* S *over* $t \approx \langle t_1, ..., t_n \rangle$ *than* another $M' = \langle N_1,...,N_m \rangle$ *over* $t \approx \langle s_1, ..., s_m \rangle$ iff the utility of M in S over $t \approx \langle t_1, ..., t_n \rangle$ > the utility of M' in S over $t \approx \langle s_1, ..., s_m \rangle$.

A sequence $M = \langle M_1,...,M_n \rangle$, of sets of constraints is *optimal* for S over t iff for some $\langle t_1, ..., t_n \rangle$ and for any $\langle s_1, ..., s_m \rangle$ no sequence $M' = \langle N_1,...,N_m \rangle$ is more beneficial to S over $t \approx \langle s_1, ..., s_m \rangle$ than M is to S over $t \approx \langle t_1, ..., t_n \rangle$.

Integration Defined

S is *maximally integrated over* t iff for each subinterval t′ of t, for every constraint C, and for all agents A_1 and A_2, if C is relevant to S relative to A_1 and A_2 over t′, then C is implemented in S between A_1 and A_2 over t′.

Integration is the satisfaction of relevant constraints by appropriate agents. That is:

S is *ideally integrated over* t iff an optimal sequence M of sets of constraints for S is implemented in S over t.

The *degree of (ideal) integration* of S over t, *int*(S,t), is

$$\max(\text{utility}(M,S,t))/\text{utility}(M^*,S,t^*),$$

where $M = \langle M_1,...,M_n \rangle$ and t Å t = $\langle t_1, ..., t_n \rangle$, for all *n*, and M^* is optimal for S over t, and t Å t*.

A system S over t is *more* (rationally) integrated than another S′ over t′ just in case *int*(S,t) > *int*(S′,t′).

The "system-ness" in a system S over a time t = *int*(S,t).

General "Theorems"

A number of general "theorems" have been developed from the theory thus far. However, the required apparatus for deriving them formally — definitions and so forth — has not yet been constructed.

Integration implies awareness of *relevant constraints*. A system can potentially have an infinite number of relevant constraints; therefore, *perfect integration* is neither possible nor desirable. Hence, there is the notion of *optimal integration* in which the set of constraints maximizing the utility (benefit of implementing and maintaining the constraints and cost of implementing and maintaining the constraints) is implemented.

Integration is always relative to system goals; this is part of the purpose of contrasting maximal integration with ideal integration. Maximal integration, the implementation of all possible constraints, clearly does not yield a useful or illuminating notion of integration. The useful notion essentially involves the notion of *optimality*, which is defined in terms of system goals. One does not want all possible information flow in a system; only that flow which is conducive to the achievement of system goals. Compare this idea to perception in humans and other animals. Much of what it is possible to perceive in any given situation is filtered out by the brain to make room for those perceptions that are relevant to the goals of the animal in that situation.

Another notion of *optimality* comes from the energy required to maintain system attunement to relevant constraints. Change in a system or its environment causes change in the system's constraints. This change brings about a corresponding increase in the *energy* required to maintain system attunement to relevant constraints. Thus, some expense must be incurred to *maintain* the level of integration in a system.

FRAMEWORKS THRUST

Introduction

The goal of the Frameworks Thrust was to explore frameworks supporting the development and evolution of CE systems and environments, particularly those which use information as the key enabler for integration among CE activities and participants. Such environments are characterized as EIISs. Frameworks describe, in general terms, the processes, methods, rules, and results for each stage of the evolution process. They generally apply to any type of organization. However, it is important that guidelines assist users in specializing general frameworks to meet the needs of a particular site.

One of the biggest challenges of this thrust was to identify the key characteristics and elements of such a framework because there are many views of frameworks. Isolating and defining framework concepts was the first step of this effort. With the foundation from this definitional work, the IICE team explored the development of extensions to existing frameworks for enterprise information system development. The development of more narrowly focused frameworks was explored in response to input from system developers asked to identify the most pressing framework needs for information-integrated CE environments.

Significant Accomplishments

The key accomplishments of the Frameworks Thrust are listed below.

1. Produced a knowledge-acquisition tool to collect data for analyzing the frameworks concept of systems development. The tool uses the Zachman framework, described in this report. The tool allows users to characterize the situations involved in system development and each situation's purpose, primary role types, artifacts, tools, rules, constraints, related situations, and so forth. A commercial product based on the IICE framework data-collection tool was later developed and sold by a start-up company, independent of the IICE contractor team[28].

2. Assisted the Experimental Tools Thrust by developing a framework processor. The framework processor uses an IDEF3 process description of a given system development process to prescribe workflow. Artifact management, access to tools, data manipulation privileges, and so forth are thereby controlled in a model-driven environment for system

---

[28] Structure™ is a product of the Framework Software, Inc. based in Redondo Beach, California.

development (See discussion of IDSE Level 1 environment in the Experimental Tools thrust section).

3.   Assisted the Computer-Aided Acquisition and Logistics Support (CALS) CE Frameworks Task Group to produce "A Framework for Concurrent Engineering," published through the DoD CALS office (CALS/CE Industry Steering Group [ISG], 1991). The Frameworks Task Group was commissioned by the DoD Director of CALS to characterize "best practice" in evolving an organization's information management mechanisms to support CE practice.

4.   Assisted the IDEF Users Group in producing the IDEF Users Group Framework Document (IDEF Users Group [IDEFUG], 1992). The IDEF Framework Document describes the key contributions of the Zachman framework and its relation to the IDEF methods.

5.   Developed a high-level framework for object-oriented system development, implementation, and evolution. This framework details the activities involved in object-oriented system development and identifies the methods (IDEF methods as well as commercial offerings) that would be appropriate to support each activity.

6.   Developed a high-level process reengineering framework together with a storyboard of its application with the Zachman framework.

7.   Isolated, classified, and refined various framework types; developed distinguishing characterizations of the terminology, concepts, and procedures for framework development.

8.   Published and presented papers on the subject of frameworks and framework-related issues at the AUTOFACT and IDEF Users Group conferences.

What is a Framework?

The term *framework* refers to a basic structure, arrangement, or system; thus, it is used sometimes as a synonym for the term *system*. More commonly, however, *framework* refers to a structure that organizes something. Much of the confusion about the term framework comes from the lack of a standardized definition. However, the term only has meaning with respect to the parts that participate in the structure, arrangement, or system. For example, the airframe components of an aircraft can be referred to as the framework for that aircraft, and the steel supporting structure of an office building can be referred to as the framework for that building. In physical domains, such as the ones just described, the notion of a framework is relatively straightforward. But when the parts in the structure are conceptual rather than physical, the notion of a framework generally shifts to one of an organizing structure for those parts. A key organizing effect of any such structure is the provision for describing (or

summarizing) characteristics about the conceptual parts and their relationships to one another. Understandably, then, *architecture*, a similar term, is often confused with *framework*. *Architecture* refers to a unifying or coherent form or structure. In the physical domain, it is rather obvious that the framework contributes to the form and the form places requirements on the framework. In the conceptual domain, the distinction is not as clear.

When *framework* refers to an organizing structure for conceptual parts, one of the four following meanings will most commonly be intended.

1.   A system development process — a roadmap describing, in general terms, the steps involved in system development and identifying the modeling methods, techniques, and tools to be used by those involved in the process.

2.   A development situation classification mechanism — a classification of development situations that occur during system development.

3.   A method classification mechanism — a structure for classifying methods or techniques according to some established criteria.

4.   An implementation architecture — a specification of the relationships between the parts of a system (e.g., in a microprocessor, the architecture specifies the organization and capacity of buses, temporary storage registers, and control elements).

Each of these notions depends on what one needs at the time. The choice depends on which conceptual parts must be organized into a framework-like structure to lend insight to the decisions to be made.

The first use of the term *framework* reflects the view of frameworks as structures for the system-development process. Under this view of frameworks, the parts being structured are the life-cycle analysis, design, implementation, maintenance, or decision-making activities associated with system development.

The second use focuses on classifying development situations. Zachman demonstrated that an effective way to identify and classify system development situations is by studying the artifacts of system development. Zachman studied the products of applying methods, tools, and techniques (e.g., models, lists, diagrams, charts) and thereby derived a classification of development situations. Development situations and associated information system design artifacts were classified in Zachman's framework with respect to role types involved in system development (perspectives) and general subjects of design decision-making (focus) (Zachman, 1986).

The third use views frameworks as mechanisms for classifying methods and techniques. For example, methods may be classified in terms of their ease of use for a particular class of users, applicability to systems development activities or phases, syntactic and semantic clarity of the method's language features, and so forth.

Method classification frameworks have been devised to assist with method selection decisions when a number of candidates are under consideration. Method classification may also be used to uncover voids in method technology.

The fourth use views frameworks as the information systems architectures themselves. Under this view, the parts being structured are components of the system itself (e.g., databases, networks, operating systems, integration utilities). Sometimes called an architectural framework, this view provides builders of a system with the tools required to implement the system. It also provides the unifying structure to ensure the individual pieces of a system fit together properly and work as a unified whole.

The first and second views of *framework* were the principal focus of the IICE Frameworks Thrust. The result of these investigations are presented below. Frameworks for method classification were investigated through the Methods Engineering Thrust. Integration architectures were explored through the Three-Schema Architecture and Experimental Tools Thrusts.

Zachman's Framework

During the initial phases of the IICE program, we investigated framework issues, basing much of our work on the information systems architecture research conducted by John Zachman. Researching the system development practice of architects and engineers, Zachman attempted to answer the question, "What is (an) Information Systems Architecture?" In his paper (Zachman, 1986), he reports finding common structural elements among the varied artifacts of the system development process. Zachman's framework (Figure 21) attempts to characterize the situations that require these representations. His characterization also attempts to identify the roles, responsibilities, conditions, prior commitments, and information that results in the need for a particular class of representations. One could characterize the resulting descriptive framework (Zachman, 1986) as classification by similar form among information system architecture representations. The framework is a matrix in which the five rows represent different perspectives (or role views) and the three columns represent the primary areas of focus for descriptions of the information system architecture.

Other frameworks derived from the original Zachman framework are organized in a similar way. The IDEF Users Group framework (Figure 22) extended the focus dimension of the Zachman framework and characterized additional columns with common interrogatives like why, who, when, how, what, and where. For example, development situations concerned with the rationale for information systems that support the business system are represented by cells in the Objective (WHY) column. The framework's cells can represent other descriptions of objects of interest to the business in the same manner. The rows represent various perspectives in the organization. This dimension organizes the descriptions of the system architecture with respect to multiple role viewpoints (e.g., the executive role [Concept Description], the manager role [Business Description], and the programmer role [Detailed Description]). Each cell in the matrix represents an enterprise situation with corresponding system descriptions constructed from the perspective of some role viewpoint.
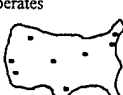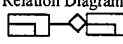
142

	DATA	FUNCTION	NETWORK
OBJECTIVES/ SCOPE	List of things important to the business ENTITY = Class of Business Thing	List of processes the business performs Process = Class of Business Activity	List of locations in which the business operates
MODEL OF THE BUSINESS	e.g., Entity / Relation Diagram ENTITY = Info. Entity RELN = Bus. Rule	e.g., Function Flow Diagram	e.g., Logistics Network Node = Bus. Unit Link = Bus. Relatn
MODEL OF THE INFORMATION SYSTEM	e.g., Data Model ENT = Data Entity RELN = Data Reln	e.g., Data Flow Diagram	e.g., Distributed System Arch. Node = Information System Function Link = Line Char.
TECHNOLOGY MODEL	e.g., Data Design ENT = Segment RELN = Pointer	e.g., Structure Chart	e.g., System Arch. Node = Hardware Link = Line Spec.
DETAILED REPRESEN- TATIONS	e.g., Data Design Description ENT = Field RELN = Address	e.g., Program	e.g., Network Architecture
FUNCTIONING SYSTEM	e.g., Data	e.g., Function	e.g., Communication

Figure 21

Zachman's Original Framework

This extension of the Zachman framework was largely motivated by the observation that all important development situations are not covered by the Zachman framework. Zachman recognized this shortcoming but resisted extending his framework beyond three columns because he could not find a sufficient number of well-established artifacts leading to the discovery of additional columns. In other words, Zachman's approach to classifying system-development situations was limited by the extent to which system-development practices had been formalized into widely practiced methods and techniques. An alternative technique for identifying development situations was obviously needed.

An alternative approach uses questions that should be answered during the system development process to help identify development situations and situation types. Rather than trying to identify development situations up front, it is sometimes easier to begin by collecting the questions that should be answered by the yet unidentified situation types. These questions can then be analyzed to identify reoccurring development situations.
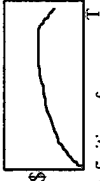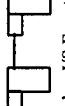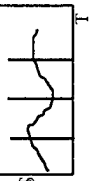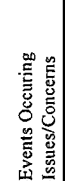
FOCUSES

PERSPECTIVES

	MANAGERIAL & HUMAN SUBSYSTEMS			TECHNOLOGICAL SUBSYSTEMS		
	BUSINESS OBJECTIVES: "WHY"	ORGANIZATION: "WHO"	LIFE CYCLE: "WHEN"	FUNCTION: "HOW"	DATA: "WHAT"	NETWORK: "WHERE"
CONCEPT DESCRIPTION	List of Business Strategies; Objective= "Acquire Business," etc.	List of Organizational Units; Organization= High Level Org. Unit	Business Life Cycle; Aggregate of Products	List of Processes the Business Performs; Process= Class of Business Process	List of things important to the Business; Entity= Class of Business Thing	List of Locations where the Business Operates; Node= Location
BUSINESS DESCRIPTION	e.g., Business Objectives to Meet Strategies	e.g., Organization Chart; Structure to control/direct tech. & people	Product(s) Life Cycle; Definition of Product Life Cycle	e.g., Function Flow Diagram; Process= Business Process I/O= Business Resources	e.g., Entity Relation-ship Diagram; Entity= Business Entity Relationship= Business Rule	e.g., WATS MCI; Node= Business Unit Link= Business Relationship
SYSTEM DESCRIPTION	e.g., Definition of Measurements	e.g., Organization Structure; -Job Roles & Relationships -Communications & Information	e.g., Process Cycle	e.g., Decomposition of Flow Diagram	e.g., Data Model; Entity= Data Entity Relationship= Business Rule	e.g., System Architecture; Node= I/S Function (processor, storage access, etc.) Link= Line Characteristics
TECHNOLOGY CONSTRAINED DESCRIPTION	e.g., Definition of Collection Methods	e.g., Work Group; -Willingness to accept change -Work Group factors	e.g., Definition of Life Cycle Span	e.g., Further Decompositions of Flow Models	e.g., Data Design; Entity= Segment/Row Relationship= Pointer/Key	e.g., Hardware/ System Software; Entity= Hardware/ System Software Line= Line Specifications
DETAIL DESCRIPTION	e.g., Measures	e.g., Work Group; -Job Satisfaction -Motivation	e.g., Specific Timeframe Description; -Events Occuring -Issues/Concerns	e.g., the Basic Work Unit; the Logical Des-cription, the "Primitive"	e.g., Database Description; Entity= Fields Relation= Addresses	e.g., the Basic Work Unit; the Physical Description
ACTUAL SYSTEM	e.g., Goals	e.g., People	e.g., Time	e.g., Function	e.g., Data	e.g., Communications

Figure 22

IDEF Users Group Framework

System development process models, project role types, project files and existing frameworks provide clues that can be used to generate questions and classify general question types. From each question type, general question templates can be developed. Representative question types and question templates have been identified in

Zachman's original paper and the IDEF Users Group Framework Document (Zachman, 1986; IDEFUG, 1992). One such question type is the *introspective* question. Introspective questions are designed to stimulate thought about factors that are considered when individuals assume different roles and perspectives during system development. A general template for introspective questions based on the Zachman framework is "<what, who, where, when> <to be verb form> the <column focus> of the <row perspective> of the <system name>?" Using a similar question template, we might construct a question like, "What are the objectives of the enterprise that are affected by the system?" Such a question would be associated with the cell in row one and column one of the Zachman framework (Figure 21).

Once question templates have been developed, their patterns can be analyzed to uncover candidate development situation classifications. The task of development situation characterization can proceed by identifying participating objects, roles, relationships, and processes. Role types are then established by identifying the individuals responsible for getting the answer or for possessing the information the question is requesting. Objects and object relations often represent elements of the answer to the question. Once the development situation descriptions have been formulated, the next step is to specify the rules that govern the question-answering process across development situations.

Application of Framework Concepts

General frameworks, like the Zachman framework, provide the basic foundations for exploring notions of the framework concept (e.g., as a management tool for systems development). In doing so, it is useful to think of the frameworks as a projection of framework information. The Zachman framework uses a two-dimensional matrix to provide this view although other formats could have been chosen. Using the matrix format, framework information is classified in terms of development situations identified by the intersection of columns and rows (i.e., cells). For each development situation, some of the framework information that may be indexed is displayed. For example, cells in the Zachman framework list examples or icons representing artifacts of information system development (See Figure 21). Additional information that can be indexed by each development situation cannot be displayed so easily.

Figure 23 provides a BOM view of the framework that displays additional information which would be required to use frameworks, like the Zachman framework, as a management tool for system development. Additional elements of framework information include cell definitions, system development process descriptions, and component methods and tools supporting development situations.
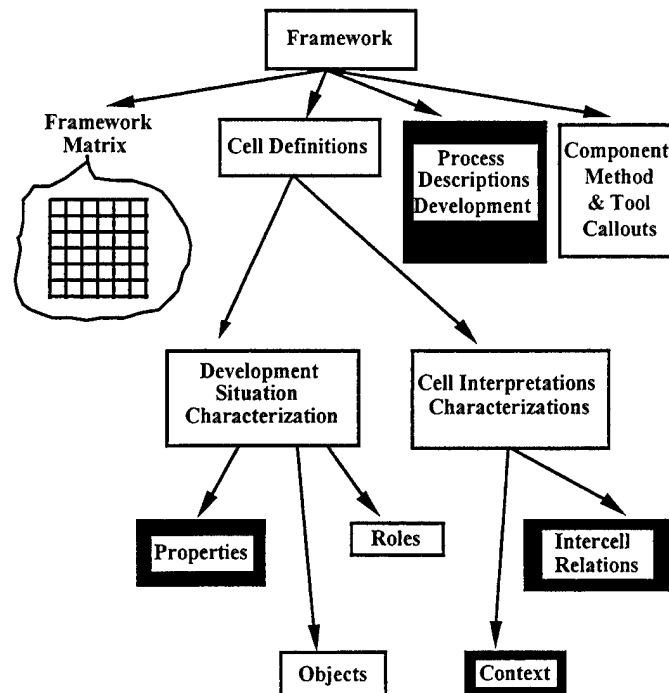
Figure 23

Framework Bill of Materials (BOM)

Use of the framework as a management tool for system development would imply defining each of these elements. Cell definitions would include, for example, listing the types of questions that should be answered in each cell and defining the mechanisms and contexts for information exchange among cells. The development process would also be described at the cell level and between cells, as would the methods and tools required to support those processes. Defining the information organized by this expanded view of the framework gives added structure to the CE evolutionary development process while also accomplishing the following:

1. Provides a "big picture" of the system-development process.

2. Provides a "quick roadmap" of the system-development process.

3. Identifies standard methods and tools to be used.

4. Assists in evaluating and selecting new tools and methods.

5. Assists in planning and scheduling the CE system-development process.

6. Orchestrates the integrated use of methods and tools.

Cell Definition

As stated earlier, each cell in a framework represents a characterization of reoccurring *development situations* in an organization. For example, the development situation illustrated in Figure 24 highlights an

146

executive role view while defining the organizational goals the information system must support. This development situation may include a group of corporate managers working with business planners and information systems managers to establish objectives, critical success factors (CSFs), and performance measures.

Roles:
      Corporate Managers
      System Managers
      Business Planners

Objects of Focus:
      Critical Success Factors (CSFs)
      Performance Measures
      Business Area Definition
      Organizational Objectives
      Products Produced
      Mission Statement
      Type of Business

Relations / Properties of Interest:

    CSFs ⟷ **Performance Measures**

    Mission Statements ⟷ **Type of Business**

    Organizational Objectives ⟷ **CSFs**

Data

Objectives

Figure 24

Example Elements of a Framework Cell

Several elements are needed to fully characterize the development situation defined by the cell. Individuals involved in the situation and the context of occurrence constitute one of these elements. Items or objects that appear, are used, or are produced in that situation should also be noted. Object descriptions should include important *properties* of the object and *relations* in which the object participates. Relations/properties of interest specify rules for how the definitions of this cell must correspond to those that will be produced in later stages of system development. An example of an object that might appear in this development situation would be the corporate *mission statement*. This object might be used to derive business unit objectives which then might be

147

used to identify specific CSFs for that business unit. In other words, there is likely to be some process of discovery, analysis, formulation, or decision-making that is associated with the development situation. Thus, a description of the process (or processes) that occurs during an instance of that development situation is also important to capture.

The development situation description should also include information about relationships between that situation and other situations. That is, the context of occurrence of a situation is considered an important part of the situation description. This means that to understand a particular situation, we need to know its context of occurrence. It is this context that allows a particular development situation to have meaning relative to a system development process.

Method Selection

In addition to characterizing the individual development situations (i.e., cells) which compose the framework, the methods and tools that are most appropriate for those development situations must be defined. Several considerations apply when making those determinations. The most obvious of these is the need to select methods and tools that support the discovery, analysis, design, or decision-making processes involved in the development situation. Additional consideration must be given to selecting methods and tools that support integration of effort among the activities involved in the development situation. Similarly, the methods and tools chosen should support integration of effort across development situations.

Support for maximized integration of effort is measured in terms of procedural complementation (e.g., minimized duplication of effort, smooth transitioning between tasks) and data sharing accommodated by a given set of methods and tools. One way methods and tools can work to support integration of effort is by providing graphical representations structured to allow easy checking *across representations* of the key decisions. This correspondence-checking across architectural representations can be illustrated in the building architecture domain. Figure 25 illustrates this correspondence between the initial bubble diagrams used to determine critical connectivity relations between major functional areas of a building and space layout diagrams used later in the system-development process to represent initial space allocation decisions. One could, for example, choose an alternative technique to represent initial space allocation decisions (e.g., a bar chart or a pie chart). However, the space layout diagram makes it easy to check conformance with the connectivity constraints displayed in the first diagram.

Bubble diagrams are used for requirements definition whereas the space allocation diagrams are used for an entirely different development situation involving design activities. Conceptually, these two development situations are represented by different framework cells related in terms of constraints made explicit by their respective diagrams. Different methods, different graphical representations, and different information were needed to support the needs of each development situation adequately. The transition between these development situations was made easier by considering how to maximize integration of effort within cells and across the cells. Figure 26

illustrates how similar constraints among cells in the framework can be identified and carried forward through the development process. Methods and tools that explicitly support the maintenance of these constraints facilitate efficient movement through the development process. Thus, relationships established through cell definition help to define the integration requirements of the methods and tools called for in the cells of such a framework. However, one should keep in mind that these integration requirements may vary from site to site depending on the needs and rules governing system development at that site.
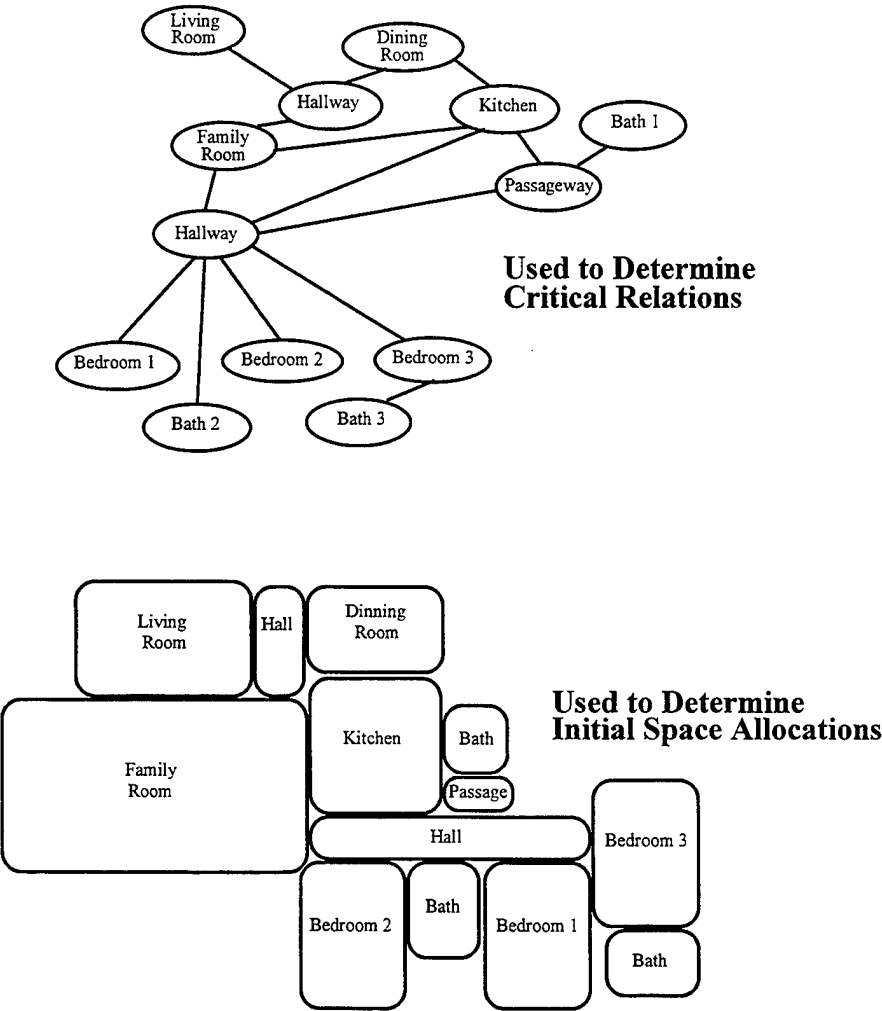
Figure 25

Interlocking Hierarchy of Architecture Representations
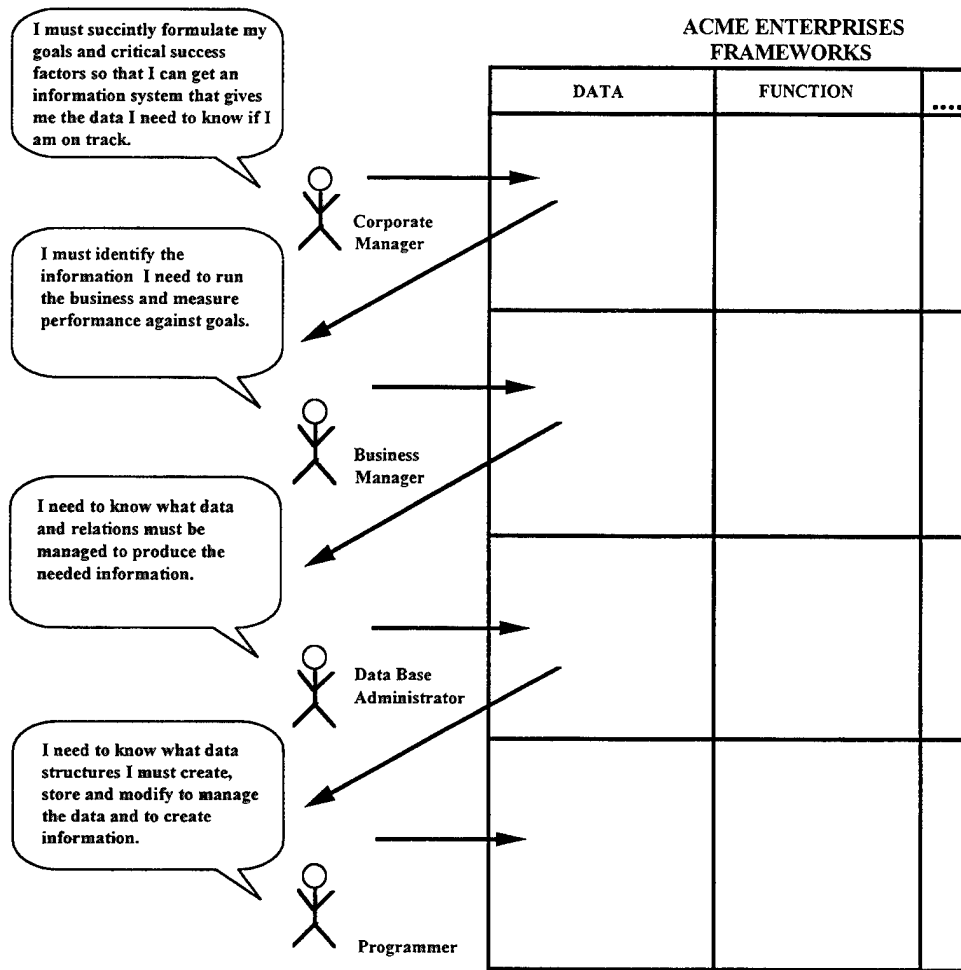
Figure 26

Intercell Integration Facilitated by Selection of Complementary Methods

System-Development Process Definition

The ordering of rows in the Zachman framework may lead one to presume that a predefined system-development process is implied. A natural ordering of activities would begin in the top, leftmost cell of the framework and work left-to-right to complete all the tasks implied by the cells in the top row. The process could then proceed by moving to the next row, following the same pattern until each cell of the framework has been "visited," and the development situations represented thereby have been completed. This would constitute a top-down, data-driven approach based on a waterfall model of development. Several alternative approaches can be considered. Each may be viewed as distinct paths traversing development situations (cells) according to some predefined system development process. Figure 27 illustrates a high-level traversal across various development situations like a thread linking various cells of the framework matrix. In this figure, some cells are "visited" while others are ignored. It is likely that the set of "visited" cells and their sequencing will be different for different

classes of system-development activities at a site. Therefore, when a site-specific framework is defined, an overall system-development process description must be specified.
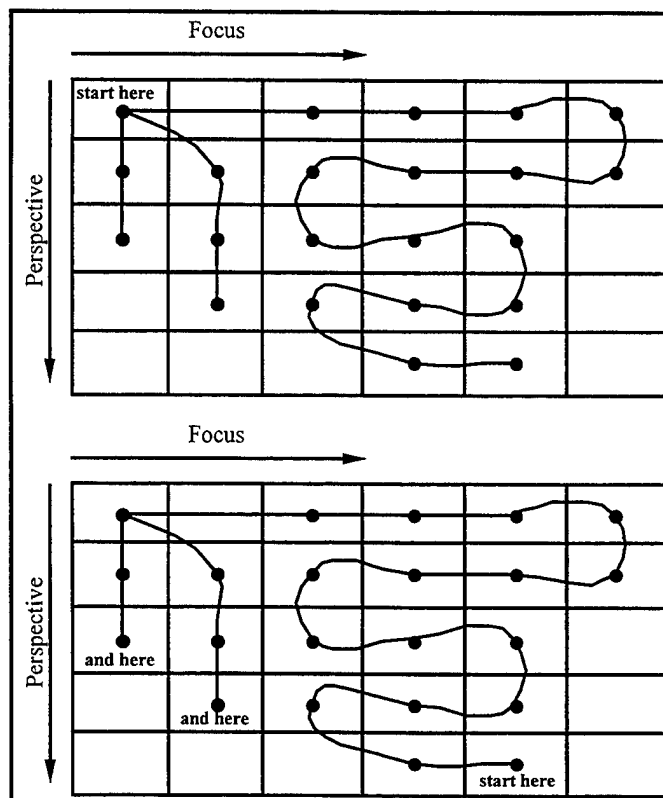
Figure 27

Possible Traversals of Development Situations in a Process

In most situations, a generic framework of the overall system-development process will have loose sequence constraints on the cells. Specified sequencing will arise from the existence of constraints (mostly between objects that are shared between cells). Where only loose ordering constraints are required, the IDEFØ Function Modeling method can describe the system well. However, in many cases, it will be important to define sequence relations. This temporal element may be defined using the IDEF3 Process Description Capture Method. With IDEF3, each development situation in the framework matrix can be represented as UOBs at the overall system-development-process level. Sequencing and relative timing relations between these situations can be represented in the IDEF3 process description schematics. The multiple decomposition capability of IDEF3 supports the definition of multiple role views of higher-level processes.

Process definition is also needed at the cell level in addition to the overall system level. The cell level of process definition describes the specific activities of each role that participates in the development situations indicated by the individual cells of the framework matrix. One component of this information is a process description that identifies the activities associated with that cell. These system development process definitions also

include descriptions of the procedures for analysis, decision-making, and configuration control; call-outs for the application of specific methods; definitions of common information/data across the different methods; descriptions of how the results of method application will be used; and role definitions.

Issues in Tailoring Frameworks for Site-Specific Use

General frameworks such as (Zachman, 1986) and (IDEFUG, 1992) are useful as reference documents. However, to support enterprise integration, the cell descriptions of these reference frameworks need to be specialized for a particular organization. These specialized frameworks are known as *site-specific frameworks.*

At a superficial level, site-specific framework definition appears to be a trivial task; however, three issues complicate the process. First, only experienced system developers have knowledge of the important recurring situations and complex interactions between roles and objects in the domain. Second, accurately describing each such situation so that it is clearly understood by others involved in the development process is nontrivial. Finally, generic frameworks are excellent starting points but do not fit specific situations.

Our experience in defining generic frameworks and site-specific frameworks indicates that the cell definition process is a highly iterative activity; in fact, it is a type of knowledge acquisition. Typical situations that drive the iterations include the following:

1. The need to refine the assignment of perspectives to rows of the framework matrix. Deciding which role perspectives can be grouped together and which need their own row requires several iterations.

2. The need to refine the assignment of focus topics to columns of the framework matrix. The original Zachman framework, for example, includes three columns labelled with the focus topics data, function, and network. The tendency is to assume a focus topic of a column will apply consistently across all rows when, in fact, the information relevant to a focus column may apply only to a subset of the perspectives represented by the rows.

3. A lack of a clear characterization of the objects or relations between objects in a development situation. For example, developing a description of a CSF is or clarifying phrases like "align performance measures with business goals" can be a formidable task.

Summary

The intuitive notion of a framework as a standard guide to successful system development is appealing. The search for such a framework has motivated the development of many offerings that continue to compete for acceptance and use as standards. However, the lack of standard definitions of framework concepts, and indeed of the term *framework* itself, has slowed the progress of these efforts. Different yet complementary views of what a

framework should be result in frameworks typically emphasizing development situation classification, system-development-process definition, method and tool classification, implementation architecture specification, or some combination of these.

Each of these views is important to system development. Furthermore, it is useful to view alternative graphical depictions of a framework as simply that; that is, two-dimensional matrices, three-dimensional matrices, charts, and so forth are simply views imposed on some set of framework information. These graphical depictions provide useful perspectives on framework information and serve as a quick index to that information. However, they do not directly display all the information needed to use frameworks as a system-development management tool.

Using framework concepts to help manage system development requires the characterization of development situation types, the definition of framework- and cell-level processes, and the selection and allocation of methods and tools supporting maximized integration of effort. General frameworks, like the Zachman framework, provide the foundations by identifying many of the important development situations that must be considered. However, the generic framework must be specialized by identifying participating objects and object types, roles, relationships, and processes that will be needed for the site.

For extensive use of frameworks as a system development management tool to be practical, additional work is needed to explore alternative views of framework information and to characterize fully the range of development situations that must be considered during system development. Some of the most promising results in frameworks research emerged through analyzing artifacts of the system development process (e.g., diagrams, models, charts, lists) to uncover important system-development situations. Additional development situations known to exist yet not evidenced by well-established artifacts indicate that large gaps still exist in our science of method technology. However, alternative approaches to framework definition have helped fill some of these gaps by uncovering additional method and tool needs, some of which have been addressed through the Methods Engineering Thrust.

This page intentionally left blank.

THREE-SCHEMA ARCHITECTURE THRUST

This section presents a brief introduction to the Three-Schema Architecture (TSA) Thrust, followed by a more detailed description of the concepts and issues driving TSA research.

Introduction

The TSA Thrust explores concepts surrounding the TSA. The point of this exploration is to incorporate the three-schema concepts into the architectures of the IDSE (see Experimental Tools Thrust) and to explore practical applications of the IST.

Goals

The primary goal is to develop three-schema based architectural design principles consistent with the IST to enable the development of EIISs for CE applications.

Specific Objectives

The specific objectives of the TSA Thrust were to:

1.   Analyze the ISO TSA concepts for applicability to EIIS development.

2.   Analyze existing conceptual schema (CS) models used in other projects for applicability to EIIS development.

3.   Establish guidelines for conceptual schema development.

4.   Establish guidelines for architecture design for conceptual schema applications.

5.   Establish metrics for conceptual schema languages.

The IICE thrusts for IST, Experimental Tools, TSA, Ontology, Frameworks, and Methods Engineering are all related and depend on one another for exchange of information and feedback. Figure 28 shows the IICE project from the viewpoint of the TSA Thrust. The points of the star show the contributions of the TSA Thrust to the other thrusts; the arrows show what is provided to the TSA by the other thrusts.

At the top of the star is the Frameworks Thrust; the foundations of this thrust were provided by research in the TSA area. The Framework effort used many principles from the TSA concept to separate an information system into different perspectives and focuses. The framework itself helped determine the voids in the ISO TSA.

Frameworks
Thrust

voids in
conceptual
schema

tools for
using and
supporting

conceptual
schema
language

Experimental
Tools Thrust

foundations

Ontology
Thrust

information
system
focus

**Three-
Schema
Architecture
Thrust**

information
system
focus

feedbacks
from info
systems

information
system
focus

theoretical
foundations

supporting
methods

Integrated
Systems
Theory Thrust

Methods
Engineering
Thrust

Figure 28

The Three-Schema Architecture Thrust Perspective

Moving clockwise around the star, one next comes to the Ontology Thrust. TSA provides an information system focus to the ontology work. Ontology capture requires a vast amount of data to be useful; this requirement implies that information system support is a must to effectively and efficiently manage the ontology data. An automated system for collecting and maintaining the ontology is required. Ontology capture efforts exemplify the challenges inherent in constructing and maintaining large distributed information systems. The primary contribution of this thrust is the use of an ontology language as the primary conceptual schema language. The use of an ontology language is discussed in depth in the section on Experimental Tools; however, essentially, an ontology can contain the information to represent the schema in the database. This use of an ontology language allows the schema and instances of the data to be stored and retrieved similarly. This storage and retrieval capability, when combined with the ISyCL capability to describe constraints, results in the genesis of what could be a functional conceptual schema.

The Methods Engineering Thrust provides the supporting methods technology to aid designers and developers in constructing TSA systems. Included in these methods are the process- and ontology-capturing

156

techniques which have been lacking in previous systems. Using structured methods with formal procedures will help create these TSA systems.

The IST provides the theories of information rooted in first-order logic, situation theory, and set theory which lend formal mathematical soundness to the work. Finally, the Experimental Tools Thrust provides the tools and automation that support the design, development, implementation, and maintenance of large distributed information systems. These thrusts work together to help unravel the difficult and complex challenge of providing today's organizations with flexible, cost-effective, reliable, distributed information systems.

Significant Accomplishments

The accomplishments of the TSA Thrust are summarized below.

1.   Surveyed and analyzed 15 engineering- and systems-related database information systems that were based on the three-schema concept (see Table 7).

2.   Researched and analyzed the ISO and American National Standards Institute/Standards Planning and Requirements Committee (ANSI/SPARC) documents related to TSAs and conceptual schemas in particular.(see American National Standards Institute [ANSI], 1975; ISO, 1982; ISO, 1987; and van Griethugen, 1982).

3.   Used the lessons learned from the above-mentioned research in the design, development, and implementation of Levels 1 through 3 of the IDSE, a distributed environment supporting data and function integration, artifact management, and work group management.

4.   Significantly impacted the design and development of the ESD concept of the Experimental Tools Thrust.

The results from the TSA Thrust has supported the other thrusts directly or indirectly throughout the IICE effort.

Overall Philosophy

TSA has been proposed for many years as the ideal architecture for an information system, and it has undergone many changes since its inception in the 1970s. The current ISO standard for the three-schema model is the starting point in many projects to build systems with good characteristics regarding extensibility, maintainability, security, integrity, and so forth. This section gives a brief introduction to TSA in general and an overview of the TSA Thrust of the IICE project.

Introduction

There does not seem to be a consensus of what a conceptual schema entails. Some sources define a conceptual schema as a unique central description of the various information contents that may be in a database. In this sense, a conceptual schema is a view of the database that contains all the information in a database. Along with the data, the conceptual schema contains the description of the actions that are permitted on the data. The database can be implemented in many possible ways. Thus, object-oriented, relational, hierarchical, or network structures may be used to store the data. The contents of the database may be viewed in many ways as well. These views are represented by external schema which are derived from the conceptual schema. The physical storage structure of the data is described by an internal schema. From this comes TSA.

Not every information system can be neatly matched to the TSA framework; however, this framework seems to fit a large number of systems reasonably well. The architecture is divided into three levels: internal, external, and conceptual (Figure 29). The internal level is the one closest to the actual physical storage (i.e., the one concerned with the way the data is stored). The actual data may be stored in many different physical locations as long as the internal schema is able to access these storage areas. The external level is the one closest to the users (i.e., the one concerned with the way the data is viewed by individual users or by applications). Finally, the conceptual level, which is a "level of indirection" between the other two, may be thought of as a community user view, a view that includes the database in its entirety. The lines linking the various levels represent the mappings that govern how a piece of data goes from the external level to the internal level and vice versa. The three levels are described below.

1. The conceptual schema level describes the information contents of the database along with the rules and constraints for actions that are permissible on the data.

2. The external level describes the various views needed by users or applications.

3. The internal level describes the storage structures of the data kept in the database.

Currently, most DBMSs tend to follow the ANSI/SPARC view of a TSA fairly closely. They do not use a conceptual schema to represent the rules and constraints for actions that are permitted on the data. The architecture of the DBMS is centered on the relational database model where the application makes transactions against the relational database that has a completely defined, and well-behaved, database schema. This paradigm works well for a single DBMS; the challenge appears when this architecture is applied to the whole enterprise integrated information system. This view is not adequate for the whole enterprise. The ISO document was an attempt to introduce TSA enterprise-wide and incorporate the constraints and business rules into the conceptual schema rather than continue trying to enforce these rules and constraints through the application programs. There are many slightly different pictures of the TSA, but Figure 29 shows the basic common structure. The figure is not in complete agreement with the ANSI/SPARC study group on database management because the rules and constraints are enforced by the application program and are not considered an integral part of the conceptual schema; however, it does give a good general view of how most databases can be modeled with respect to TSA.

External Level
(individual user view)

Conceptual Level
(community user view)

Internal Level
(storage view)
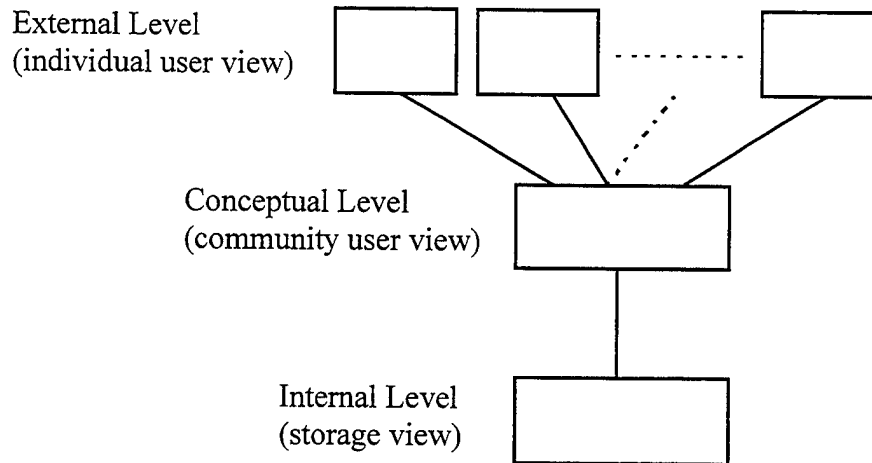
Figure 29

Three Levels of the Architecture (Date, 1990)

The ANSI/SPARC study group report has a more detailed explanation of TSA, describing various individuals' roles, showing models of typical database systems, outlining data integrity and security issues, and so forth. Another important report on the conceptual schema is *Concepts and Terminology for the Conceptual Schema and the Information Base* (ISO, 1982), which goes into more detail about the dynamic aspects of the conceptual schema. These documents form the basis for the functionality that a TSA system would support in an ideal situation. For more in-depth treatment, refer to literature of the following authors: Date (1990), ANSI (1975), ISO (1982), ISO (1987), and van Griethugen (1982).

From these sources, we derived the six specific requirements and four roles of a conceptual schema shown in Tables 5 and 6.

Table 5. Requirements for a Conceptual Schema

1.	Must adequately describe both the static and dynamic aspects of a universe of discourse.
2.	Must provide a language with which a conceptual schema can be expressed such that it is understandable to a user.
3.	Must provide a language which communicates the conceptual schema to a computer.
4.	Must provide for ease in modification to the conceptual schema.
5.	User views of the database must never contradict the conceptual schema. Changes to external schemata do not affect the conceptual schema.
6.	Must be invariant to changes in the internal representation of the data.

Table 6. Roles of the Conceptual Schema

1.	Establish a common basis for defining the general behavior of the universe of discourse.
2.	Define allowable manipulation and evolution of the information about the universe of discourse.
3.	Provide the basis for interpreting external and internal representations of the information about the universe of discourse.
4.	Provide mappings between and among external and internal schemata.

The three-schema approach in information system architecture was developed to separate the user applications from the physical representation of the structure of the data. The idea behind the three-schema approach is that it will provide the logical view of all the information managed by the enterprise. However, there are few (if any) true implementations of the three-schema architecture. Many DBMSs provide the user view/external schema (the data required by a single application) and the internal schema (the actual physical storage description) but lack the conceptual schema. There needs to be a separation of the user view of data, the enterprise view of the data, and the physical representation of the data. The conceptual schema provides this enterprise view. However, in the transfer of data from the user view to the physical realization of the data, the conceptual schema is not an intermediate state. Because most implementations appear to handle the external and internal schema descriptions reasonably well, discussion in this report is focused on the conceptual schema.

No current system actually supports a conceptual level that meets all the requirements set out by the ANSI or ISO documents. We are still trying to move toward such systems and may be able to perform comprehensive security and integrity functions from the conceptual level in the future.

As stated previously, the conceptual schema is the logical enterprise view of the data. The ISO document, *Concepts and Terminology for the Conceptual Schema and the Information Base* (1982), defines the conceptual schema as, "The description of the possible states of affairs of the universe of discourse including the classifications, rules, laws, etc., of the universe of discourse." That is, the universe of discourse is the collection of all objects/entities from a selected portion of a real-world that are of interest; the description of the specific objects that exist in a specific instant in time make up the information base.

One objective of this thrust is to inspect the state of the art in three-schema architecture implementations (those which claim to adhere to the ISO definition). From this inspection, we question how the three-schema theory has evolved since its inception in the 1970s, how information system needs have evolved since then, and what approaches will lead to a three-schema information systems implementation.

160

In the ISO view, the conceptual schema describes the types and variables that can exist, the concepts that are not subject to frequent changes, and the rules that influence the behavior of the conceptual schema and the information base.

The ISO definitions for some important concepts are as follow (van Griethugen, 1982):

> A *conceptual schema* is a consistent collection of sentences expressing the necessary propositions that hold for a universe of discourse.

> *Static aspects of a system* are those aspects of a system that apply to each of its states.

> *Dynamic aspects of a system* are those aspects which govern the evolution of a system.

> *Static rules or constraints* establish dependencies between parts of the system at any instant in time.

> *Dynamic rules or constraints* establish dependencies between parts of the system through several instances of time.

If we accept the 100-percent principle that "all relevant general, static, and dynamic aspects (i.e., all rules, laws, etc.) of the universe of discourse should be described in the conceptual schema" and if we accept the ISO definitions before a three-schema implementation can be designed, we must ask probing questions about practical issues, implementation approaches, evolution issues, and current implementation attempts.

Lack of Success Issues. Numerous attempts to construct a conceptual schema-based system have had limited or no success Why is such a seemingly robust concept so difficult to achieve? Is it wise to attempt direct implementation of the concept? Currently, there is no known set of rules which govern how the mappings from the external to the conceptual and from the conceptual to the internal schemas are to occur, *and* still ensure the conceptual schema business rules are not compromised.

Evolution Issues. Clearly, the "conceptual" elements of information systems evolve over time. For example, FORTRAN was considered at one time a conceptual-level specification because one did not have to worry about the specifics of register allocation. The programmer was able to work at a more "conceptual" level because he did not have to spend time detailing how a particular computer's internal registers were to be used. He could do his problem solving at higher conceptual level supported by the FORTRAN language and rely on the compiler to handle the low level details. Now, fourth and fifth generation languages and object-oriented analysis and design are the conceptual specification tools. In any implementation of three-schema architecture, the conceptual schema must evolve over time. In the ISO document, evolution appears to be restricted to the addition, modification, or deletion of the contents of the conceptual schema. However, it is likely that the representational language itself will evolve. For instance, consider the impact of the IICE effort. Under the IICE effort, new languages have been defined (or existing ones refined) for model description, ontology description, and (in this thrust) conceptual schema language

descriptions. However, further research and evolution will still be necessary. It is likely that languages with greater expressive power will emerge in the ten years following this effort. conceptual schemas built in these languages will probably consider the IICE situation theoretic languages to be very "procedural" and "implementation" oriented.

Conceptual Schema Language. An implementation of a conceptual schema requires a language for describing the conceptual schema to the information system. Establishing the metrics to evaluate the suitability of a conceptual schema language requires a correct determination of the role the conceptual schema ought to play and the contents of that schema. Some possible evaluation criteria are listed below.

1. How well does the conceptual schema language support the dynamic constraints that must be represented to maintain the necessary business rules of the information system?

2. How well does it work representationally to describe the concepts and objects in the domain and relate them to the information in the information system?

3. How well can it be used, understood, and manipulated by the people who must build, maintain, and use it on a regular basis?

The next section briefly describes the conceptual schema models that were reviewed in an attempt to gain insight to why the three-schema architecture is so difficult to implement.

Overview of Conceptual Schema Models Reviewed

Historical three-schema architectures do not conform to the definition of a three-schema architecture given in the ISO document. While historical systems can be cast into the ISO definitions, few have the capabilities to define the dynamic aspects and rules that govern how statements can be added to the universe. Nevertheless, this can be done — although it has not been done thus far.

As the needs of the enterprise change, the requirements of the information system must change as well. The need to change the information that is managed by the system makes the three-schema architecture very difficult to implement. Another problem is that the requirement to describe the complete universe of discourse *a priori* is not very realistic. The system will always be changing making any complete description impossible. The systems and languages that were analyzed were never matured into fully blown production systems because of performance difficulties. It takes an enormous amount of computing power to implement a full three-schema architecture-based system. More work is required in the area of defining processes and constraints. The data definition components of the conceptual schema, while not trivial, are better understood than the processes and constraints that are also facets of the conceptual schema. The tools and methods to model the data for the conceptual schema already exist. However, the constraint and process information still need tools and methods to capture,

model, and interpret them in a central conceptual schema and realize the vision set forth in the ISO three-schema architecture paradigm.

To understand the difficulties inherent in constructing three-schema-based systems and to understand the complexity of generating a conceptual schema, many systems based on the three-schema vision were examined. This marks the first attempt to define metrics for using Conceptual Schema Based Database Systems. The metrics were categorized using the following criteria:

Language or System: Denotes whether the three-schema architecture in question relates to a language for conceptual schema definition, a three-schema-based system, or both.

Conceptual Schema Definition: Denotes which conceptual schema definition for is used — the older definition or the newer (ISO) definition.

Specific or General: States whether the three-schema architecture has global significance or relates to specific areas only.

Graphic Language: Indicates whether a Graphic Representation language for conceptual schemas is used.

Processable: Indicates whether the language is processable by computer.

Conceptual Schema Requirements: States which of the six requirements for a conceptual schema are addressed (see Table 5).

Conceptual Schema Roles: Denotes which of the four conceptual schema roles are addressed (see Table 6).

All of the systems that were analyzed were research prototypes; none matured into production systems because of performance issues. In addition, there is no evidence of any fully ISO three-schema architecture-compliant systems in existence. Many systems have a three-schema architecture in the most general sense, but all fall short of a full implementation. Because of the limited success of historical three-schema architecture systems, we have tried to use the three-schema architecture design as a guide rather than a rule in designing the ESD. The separation of the static and dynamic aspects of the database from the application programs is a strong advantage of the three-schema architecture; however, it is difficult to achieve. Our approach is to model the aspects of the enterprise information system using different methods to capture these aspects. The models are combined in the ESD, which then serves as the conceptual schema and the database.

Approaches to Three-Schema Architecture Information Systems

This section describes two different approaches to developing three-schema-based information systems. The first is a brief description of an initial paradigm that has since been replaced; that is, a description of the past.

163

The second is a description of the present. It summarizes our currently held beliefs and ideas about the challenge of creating a three-schema system.

Multischema/Multimodel. Initial attempts to develop a good approach for developing three-schema-based information systems began by identifying what was missing from the three-schema architecture concept. Many business rules are still programmed into the application programs for the information systems instead of being encoded in the conceptual schema as recommended by the ISO and ANSI reports. These business rules or constraints are the main reason the three-schema architecture is so difficult to implement in its pure form. Current conceptual schema models do a good job of modeling data but are not adequate in defining data security, consistency, and integrity constraints. A multischema architecture approach extends the three-schema architecture by adding other schemas that encode "missing" business rules previously held by the application program. Figure 30 shows the sample multischema architecture originally conceived.

Figure 30

Multischema System Description Mechanism

Resolving the problems inherent in the data models is a big part of creating a working system; however, data models do not completely cover all aspects of the design. Many other factors of a dynamic nature are required for the system to be completely defined. This requirement is one main reason no one has been able to create a pure three-schema architecture that embodies the vision set forth in the ANSI/SPARC and ISO documents.

The multischema paradigm augments the conceptual schema by adding the necessary models that capture the dynamic information component of the evolving system description. In this way, the rules and constraints that once were managed by the various application programs can be collected into the various models that describe the

system. Thus, the vision of the conceptual schema containing all the static and dynamic rules can be achieved by this method. The multischema system description, in addition to the original three-schema description, contains many other schemas which determine processes, constraints, artifacts, and so forth. The multischema is really a multimodel within the conceptual schema. In other words, the conceptual schema should contain all these models (e.g., process, constraint, artifact, data, and activity models)and should use the models to determine the behaviors and contents of the information system being created. In this way, the conceptual schema is augmented to include models that describe the system from many different perspectives. The conceptual schema is more than just data models of the DBMS; it includes the processes, artifacts, constraints, integrity rules, business rules, and the like. The multischema system description includes these facets of the conceptual schema as models that can be modified and manipulated to change the behavior of the system. Thus, both the static and dynamic aspects of the conceptual schema are captured in the various types of models which separate these aspects from the programs and makes them explicit rather than implicit. We have experimented with this approach by implementing this kind of environment in the IDSE.

IDSE Approach to Conceptual Schema. The IDSE has evolved into the Level 3 system — an attempt to create an environment that implements a major portion of an ISO three-schema architecture system. Figure 31 illustrates the IDSE architecture.
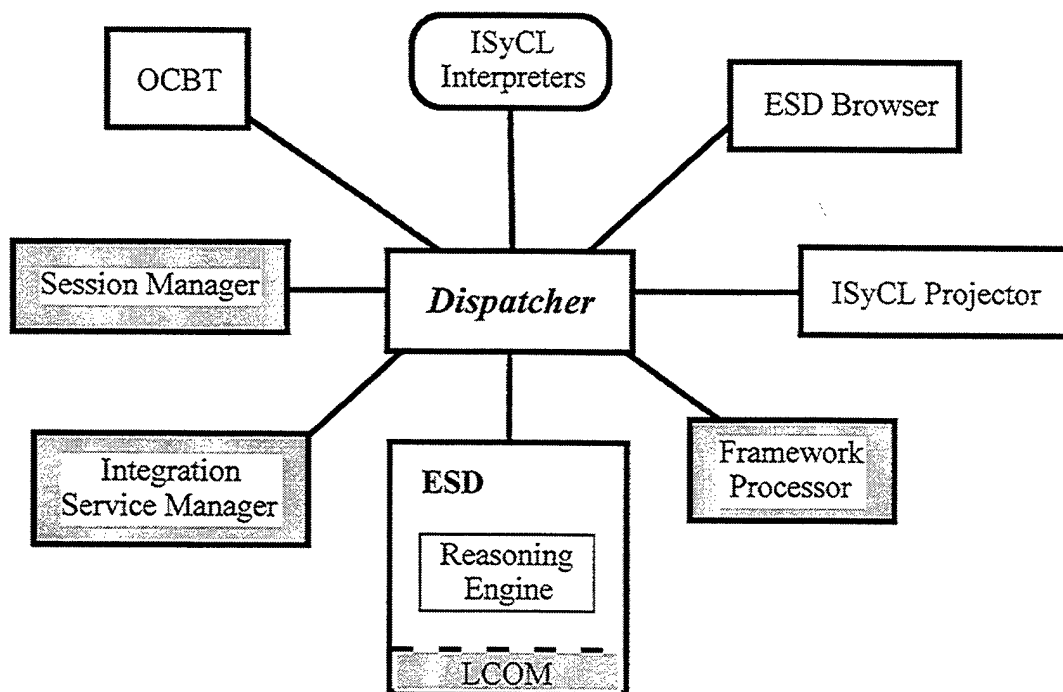
Figure 31

IDSE Architecture

165

In Level 3 of the IDSE, the ontology modeling tool and method make the constraints dynamic and user-editable. Thus, the constraints that were lacking in the analyzed systems can be captured using the ISyCL elaboration language (Fillion, 1995). Thus, with respect to three-schema architecture, the rules and constraints that were implicitly enforced by application programs can now be explicitly recorded and managed within the conceptual schema; changes will no longer need to be implemented by reprogramming the applications. ISyCL is a linear textual language with semantics to handle the complexity of describing the rules and constraints not represented in conventional graphical or textual methods. ISyCL is used to describe and elaborate on the models that make up the conceptual schema.

In addition to the use of ISyCL for constraint specification, multiple modeling methods and tools capture the various aspects of the system such as processes, data definitions, and constraints. An expressive ontology language augmented with the ISyCL constraint specifications is the conceptual schema language for the system models. The models are used to define the system and are linked by a common ontology language. The system is modifiable through the models, and the whole system can be defined at the conceptual schema level without encoding the application programs with the constraints The ESD acts as a conceptual schema and receives information about the information it contains through the ontology models. Using the Container Object System (COS) and the Constraint Propagation System (CPS), the ESD can maintain the static and dynamic components that are required for a conceptual schema to operate. The results of this approach is that the conceptual schema is stored in the ESD and can be modified by changing the models that represent the information system. The ESD corresponds directly to the conceptual schema of the three-schema architecture and contains the models that represent the many facets that make up the conceptual schema. The ESD is completely editable and can be changed without having to change the application programs that access the data. The user can browse and modify the schemas, constraints, data, and rules that are stored in the ESD through the use of the ESD browser. The ESD browser lets the user see into the ESD and examine the schemas and information they represent directly without having to go through a particular application. With the use of the ontology capture and browsing tool, the programs that will access the system and the underlying databases are registered with the ESD. This makes the whole ESD extensible to support other applications and modeling tools.

At the base of the ESD is the life-cycle object manager (LCOM) which performs the data storage and retrieval operations as they are requested by the ESD or one of its components. This roughly corresponds to the internal schema in the three-schema architecture. The OCBT is the mechanism by which the ESD is extended to include new applications and modeling tools. This is accomplished by describing the ontology of the application or tool using the OCBT. Once the ontology of the application is captured, it is registered with the ESD as a new application. This causes the ESD to direct the LCOM to create all the schemas and storage mechanisms necessary to manage information about the application in question. After registering an application, the ESD is ready to handle transactions on the data that the application will store and retrieve. The application interface to the ESD is provided by the interpreter. If the application can import and export ISyCL, an interpreter is not needed as the

interpreter functions solely as a translator to convert an application's native format into ISyCL statements. Communication is accomplished by sending messages containing ISyCL statements between the application and the ESD. The interpreter roughly corresponds to the external schema in the three-schema architecture. The reasoning engine, built around the CPS, maintains constraints and rules. The constraints are stored in the ESD (ultimately the LCOM), and the CPS constantly monitors every change to see how it will affect the rest of the system and information. In this way, we can implement the dynamic aspects that are outlined in the ISO three-schema architecture. This technique is described in more detail in the section on the IDSE in the section entitled, Experimental Tools.

Goals

The general goal of the three-schema architecture Thrust is to develop three-schema-based architectural design principles consistent with the IST to enable the development of EIISs for CE Applications.

The goal of the three-schema architecture Thrust is to produce a set of guidelines for conceptual schema and information system architecture development. Additional goals are to identify and establish metrics to evaluate conceptual schema languages, and to support the other IICE thrusts by facilitating and enabling information systems development using the three-schema approach and principles.

Accomplishments of the Three-Schema Architecture Thrust

This section describes some accomplishments of the three-schema architecture Thrust.

Survey of Three-Schema Information Systems

To gain an understanding of the problems that arise in the development of three-schema architecture, we have researched and analyzed other projects that have used the three-schema architecture model in the development of their information systems. These historical systems, taken from different engineering arenas are summarized in Table 7.

Table 7. Summary of Conceptual Schema Models Reviewed

Projects	
CFI	**CAD Framework Initiative (CFI, 1991)**
Area	CAD Systems
Focus	Integration of CAD tools, data specification standards
Objective	To access to CAD data from various CAD tools, standards.
DICE	**DARPA Initiative in Concurrent Engineering (Reddy, 1989)**
Area	CE

Table 7. (Continued)

Focus	Integrated Engineering/Manufacturing Systems
Objective	To develop a methodology to improve (formalize) the practice of CE.
Rensselaer Object System for Engineering (ROSE)	**Information Architecture for Concurrent Engineering (DICE, 1989)**
Area	CE
Focus	Information Modeling
Objective	Used in the DICE program to reduce the length of the design cycle.
ISO Computer Information Management (CIM) Architecture and Framework	**(CIM, 1988)**
Area	CIM Modeling and Systems Integration
Focus	Standards Development
Objective	To provide a means for describing enterprises in a processable description.
EIP	**Enterprise Integration Program (EIP, 1990)**
Area	Enterprise Integration
Focus	Framework for Enterprise Integration, Enterprise Information System Integration
Objective	To establish Enterprise Integration guidelines.
Integrated Maintenance Information System (IMIS)/ Content Data Model (CDM)	**Content Data Model (CDM, 1988, 1989; Earl, 1989; Earl & Gunning, 1990; IMIS, 1987)**
Area	Automated Technical Information Systems
Focus	Maintenance Data for Weapons Systems
Objective	To provide storage, distribution, and presentation of technical data, database specification, Standard Generalized Mark-up Language (SGML)
RIDL	**Reference and IDea Language (CDC, 1982)**
Area	Semantic Meaning
Focus	conceptual schema Representation
Objective	To provide a query and update, constraint definition, and process description language for defining and manipulating the conceptual schema and data for a database.

Table 7. (Continued)

IRDS	Information Resources Dictionary System (Goldfine & Konig, 1987; Burkhart, 1991)
Area	Repository that includes the complete description of an information system together with its surrounding enterprise environment.
Focus	conceptual schema Framework
Objective	To extend the existing IRDS conceptual schema framework
Carnot Project	**(Carnot, 1991; Huhns, 1992)**
Area	Interoperability of Heterogeneous Environments
Focus	Open Systems
Objective	Development of a set of software tools that will allow access to and maintain consistency of data in heterogeneous environments and serve as a bridge between older, closed-computing environments and present and future open systems.
BRC/BERM	**Business Relationship Classification/Business Entity Relationship Model (Jorgenson & Walter, 1991, 1992)**
Area	Business Modeling Knowledge Capture Mechanism
Focus	Business Concepts and Relationships
Objective	To develop a new method based on natural language.
I²S²	**Integrated Information Support System (Judson, 1985)**
Area	Information Integration
Focus	Communication, management and control, multiple computer environment, data sharing, ease of use
Objective	To build an integrated-information system
IDS	**Integrated Design Support System (Rockwell International, 1989)**
Area	Automated Technical Engineering Data System
Focus	Weapons Systems
Objective	Support weapons systems life cycle from requirements definition and design through manufacturing to deployment and logistics support. To construct a prototype to acquire, store, distribute, and perform configuration management.
EIS	**Engineering Information Systems (EIS) (Linn & Winner, 1986, 1989)**
Area	Engineering Information Exchange
Focus	Heterogeneity of hardware and software platforms, data formats, tools, site-specific policies, and interfaces

Table 7. (Concluded)

Objective	To develop a tool integration framework, tool portability, uniform design environment, exchange of design information, design management and reuse.
PDES/SUMM	**Product Data Exchange using STEP/Semantic Unification Meta Model (SUMM, 1992)**
Area	Graphics
Focus	Standards
Objective	To provide a database for all product data.
KIF	**Knowledge Interchange Format (Genesereth & Fikes, 1992; Gruber, 1992, 1993)**
Area	Knowledge Bases
Focus	Standards
Objective	To develop a language suitable for representing all kinds of knowledge so that knowledge can easily be transferred among different systems.

From these studies several deficiencies became apparent in the three-schema architecture. Some reasons that many efforts are struggling in their attempts to implement the ISO architecture are listed below.

1. The traditional three-schema architecture approach does not have an adequate method for specifying constraints on the "universe of discourse."

2. Application programs currently are used to encode the dynamic types of constraints which should properly be in the conceptual schema.

3. Static data specification languages are not sufficient.

4. Data definition languages (DDLs) and data manipulation languages (DMLs) do not completely define the requirements for an information system.

5. There is a disconnect between modeling the "universe of discourse" and implementing it in a DBMS. Furthermore, essential information is lost between modeling the world's objects and implementing the database that records the information about those objects.

6. Data modeling schemas is limited in specifying an application. Separating data and rules from an application is not simple.

7. All services provided by the integrated information system must be determined *a priori* for a given domain.

170

8.   In large enterprises, it is difficult to establish a common data standard throughout the organization.

These projects have tried to build the three-schema architecture in various ways, with varying degrees of success. In the IICE project, we have attempted to emulate the best of these efforts while avoiding the pitfalls they have uncovered in implementing the three-schema architecture. Any guidelines for using three-schema technology must avoid the problems inherent in the three-schema architecture paradigm.

ISO and ANSI/SPARC Documents

The research of the ISO and ANSI documents has given us insights about creating an information system modeled after the three-schema architecture. This work can be measured by the success of the thrusts that were affected by three-schema concepts and principles, the identification of the requirements and roles of the conceptual schema, and the discovery of some of the problems in the three-schema architecture design.

IDSE Levels 1 through 3

A detailed discussion of IDSE Levels 1 through 3 can be found in the Experimental Tools Thrust section which provides a summary of the three-schema architecture Thrust's influence on the Experimental Tools Thrust.

Conclusions

The three-schema architecture Thrust has been a foundation for the IICE effort. IST, Experimental Tools, Ontology, Frameworks, and Methods Engineering are all built on the three-schema architecture Thrust. The three-schema architecture research has been implemented in the IDSE Level 3 prototype.

The IDSE is an evolving system environment. Changes in the information it stores and its constraints or rules are customizable to the point that new applications can be added at will by modeling the application with the OCBT and providing an ISyCL interpreter to translate the declarative statements. An application that can import and export ISyCL statements would not need an interpreter to integrate with the ESD.

We have tried to use the concepts from the ISO three-schema architecture document and the analysis of the conceptual schema languages and three-schema architecture-based systems to develop the IDSE so that it is compatible with the three-schema architecture paradigm and takes advantage of the benefits without the problems that have plagued previous attempts at building a full three-schema architecture system. Conceptual schema development for enterprise-wide information systems can only be accomplished if the conceptual schema is not required to be specified in its entirety beforehand. conceptual schema development is a matter of using good software engineering techniques, and the best methods and tools effectively. A key success factor is realizing that no one tool or method can cover the entire spectrum of the conceptual schema. The conceptual schema has many

facets, and each facet requires different tools to model. Furthermore, the conceptual schema is not a fixed point; it is constantly changing as the enterprise evolves to adapt to a different environment. As the enterprise changes, so must the information system that supports it and, too, the conceptual schema that is the heart of the information system.

As the conceptual schema takes on more responsibility for managing constraints, rules, processes, and data, the application programs must adapt to a changing paradigm. This strongly impacts the architecture and construction of the application programs. The programmers of these systems must learn to program with this evolving metaphor of a conceptual schema that does some of the work and can change its behavior by changing the constraints. Many lessons will need to be unlearned, and many new lessons will have to be discovered. The focus is moving toward function and data acquisition or application. Developing these applications is the next challenge to be faced.

TECHNOLOGY TRANSFER THRUST

The Technology Transfer/Transition Thrust focuses on providing effective technology transfer and insertion by improving education and communication about IICE technology and by demonstrating that technology in practical real-world applications. The main objectives of this thrust have been to disseminate project results of the IICE technologies; to further our understanding of ways IDSE tools can enhance efficiency, quality, and knowledge sharing; to build competitive positions in information technology in the Air Force (specifically demonstrated at Tinker AFB), and to invite feedback to the IICE efforts.

Significant Accomplishments

The major accomplishments of the Technology Transfer Thrust are summarized below:

- Prepared and published seven Air Force Technical Reports.

- Prepared 26 papers for presentation at industry conferences or publication in industry journals.

- Participated in standards development activities, and contributed to the development of the revised IDEFØ NIST FIPS, the IEEE IDEFØ formalization standard, the Society of Enterprise Engineers' (formerly the IDEF Users Group [IDEFUG]) IDL, and the PDES SUMM standard.

- Influenced the development of several commercial projects.

- Impacted and influenced several ongoing government projects.

Approach

The Technology Transfer Thrust focuses on two primary tasks: Technical Publication, and Participation in Conferences and Standards Development Activities. The technical conference participation and standards participation tasks monitor advances in areas related to this effort and identify and target high payoff technology transfer/transition mechanisms. The IDEFUG was used as a focus early in the effort as a primary technology/transfer mechanism for research performed under this effort. The IDEFUG served as a forum for the more consumable products of the IICE effort (i.e., the new IDEF methods and IDSE technology), while the more theoretical results of the IICE effort (i.e., the NIRS and IST) were targeted for the standards activities.

173

Accomplishments

The significant accomplishments of Technology Transfer Thrust tasks include publications and presentations in conferences, invited seminars and journals, and participation in standards organizations. The results of these efforts are summarized in the following sections.

Emerging Standards

The IICE project has supported in support of national and international modeling method standards activities. The earliest of these efforts, which arose from the IST Thrust, are concerned with the development of a logic-based framework (NIRS) for the representation of enterprise model information. This research led to a number of papers on the need for a standardized logic-based framework for information representation. The most important of these, "Representation, Information Flow, and the Model Integration" (Menzel, Mayer, & Sanders, 1992), was written for the first International Conference on Enterprise Modeling Technology, which was concerned with formal standards to guide the development of modeling languages. This paper appeared in the MIT Press proceedings of the conference.

Early in the course of this research, it was discovered that a parallel effort — SUMM — was being conducted by the Dictionary/Methodology Committee (DMC) of the IGES/PDES Organization, a U.S. working group (ISO TC184/SC4/WG3) within ISO. Therefore, we contacted the chief author of SUMM, Dr. James Fulton of Boeing, and worked closely with him and the DMC during the last six months of the SUMM effort. These efforts resulted in the two-volume technical report entitled, *The Semantic Unification Meta-Model* (Fulton et. al., 1991).

Currently, SUMM is among three central standards for the representation of information adopted by the ANSI X3T2 committee on data exchange. It provides the general framework for defining the syntax and semantics of modeling languages. The other two formal frameworks, which fall under the aegis of X3T2, are IRDS — based chiefly on conceptual graphs, developed under the ANSI X3H4 Committee on IRDSs — and KIF — a textual, computer-processable logic-based language developed under the ARPA-sponsored KSE. (ISyCL, developed under IICE, is based on KIF.) We have been involved in the Interlingua discussion group on the foundations of KIF. This participation brought relevant IICE-related research to bear on problems being addressed by the international standards community.

Recently, under funding from NIST and in collaboration with Dr. Fulton, we developed a formal standard for IDEFØ, including its formal syntax and semantics. This work built on IICE research on the use of formal methods in the development of modeling method technology. The document resulting from this effort was released as a NIST Internal Report. In addition, we have been directly involved in a parallel working group, sponsored by IEEE and the IDEFUG, that has been charged with developing a formal standard for IDEFØ and with revising the NIST FIPS for IDEFØ (essentially a user's guide to IDEFØ that presents the NIST formalization in an informal,

intuitive fashion). The NIST formalization has been used as the basis of the IEEE formalization, and the document is due to be put to an IEEE ballot in June or July of 1996. Pending approval by the balloting committee, the document will be adopted as the national IEEE formal standard for IDEFØ. The document will then be submitted to ISO as the IDEFØ international standard.

Finally, there is a desire in the IDEF community and NIST toward standardizing IDEF3. (Currently, however, no plan has actually been implemented. The issue will be taken up at the next Society for Enterprise Engineers Meeting in Orlando, Florida, in June 1995.) Since the bulk of IDEF3 development occurred under the IICE program, IICE research on the method will serve as the foundation of the standard that emerges.

Another effort in regard to standards activities dealt with our participation in the IDEFUG and the IICE method integration research efforts. The IDL is a standard designed to be used by IDEF software vendors as an exchange mechanism for IDEF models. IDL allows both the syntactic and semantic information in an IDEF model to be transferred. Because of its flexibility to represent both types of information, IDL can also be used as a data interface to non-IDEF software. IDL for IDEFØ was developed through several versions before final approval was given to Version 2.2.7. The approval and testing of the final version of IDL for IDEFØ was held during the October 1993 IDEF Users Group Conference. Following the approval of the IDEFØ-IDL, all participating vendors announced planned future support.

Technical Literature

The most successful aspect of the Technology Transfer Thrust has been the technical publications achieved during the IICE effort. These publications have been accomplished through three channels: Air Force publications, conference presentations and proceedings, and industry journals. The articles and papers published under the IICE effort are listed below.

Air Force Publications. During the course of the IICE effort, reports and documents were developed to document the results of the IICE development efforts. Several of these reports were, or soon will be, published by the Air Force. The title and brief annotation of each report are listed below.

IDEF3 Process Description Capture Method Report: a description of the syntax and procedure for applying the IDEF3 method (Mayer, R. J., Menzel, C., Painter, M. K., deWitte, P. S., Blinn, T. & Benjamin, P., 1992a).

IDEF4 Object-Oriented Design Method Report: a description of the syntax and procedure for applying the IDEF4 method (Mayer, R. J., Menzel, C., Painter, M. K., deWitte, P. S., Blinn, T. & Benjamin, P., 1992).

IDEF4/C++ Object-Oriented Design Method Report: a description of the syntax and procedure for applying the IDEF4/C++ method (Mayer, R. J., Keen, A. A., Browne, D. C., Harrington, S., Marshall, C., Painter, M. K., Schafrik, F., Huang, J. C., Wells, M. S., & Hisesh, H., 1995b).

IDEF5 Ontology Capture Method Report: an overview of ontology concepts and a description of the syntax and procedure for applying the IDEF5 method (Mayer, R. J., Benjamin, P., Menzel, C., Fillion, F., deWitte, P. S., Futrell, M. T., & Lingineni, M., 1994).

IDEF Compendium: a summary of the research results to date on the development of IDEF6 Design Rationale Capture Method, IDEF8 Human System Interaction Method, IDEF9 Business Constraint Discovery Method, and IDEF12 Network Design Method (Mayer, R. J., Crump, J. W. IV, Fernandez, R., Keen, A., & Painter, M. K., 1995c pending).

IICE Technology Transition Effort for E-3 Programmed Depot Maintenance (PDM) Final Report: a summary of the results of IICE technology insertion at the Oklahoma City Air Logistics Center (Painter, M. K., Graul, M., & Marshall, C., 1995, pending).

Conference Papers and Journal Articles

Another successful technology transfer activity has been the presentation of conference papers and the publication of journal articles. A complete listing of published papers follows below. These papers typically have presented results of IICE research or have demonstrated the application of IICE technology to various systems engineering problems.

IICE: Program Foundations and Philosophy presented at the IDEF Users Group and published in proceedings (Painter, 1991).

Automated Design of Factory Simulation Models from IDEF3 Process Flow Descriptions presented at the IDEF Users Group Conference and published in the proceedings of that conference (Blinn, Mayer, & Joshi, 1991).

CIM: Modeling with an IDEF Perspective published in the CIM Implementation Guide (Painter, 1991).

Application of IDEF3: A Process Flow Modeling Method presented at the IDEF Users Group Conference and published in the proceedings of that conference (Cullinane, Mayer, & McCollum, 1991).

The Importance of Mathematical Formalization for Advancing Modeling Methods Technology presented at the IDEF Users Group Conference and published in the proceedings of that conference (Menzel, 1991).

A Roadmap for Enterprise Integration presented at the AutoFact '91 Conference and published in the proceedings of that conference (Mayer, 1991).

The Many Faces of Concurrent Engineering presented at the AutoFact '91 Conference and published in the proceedings of that conference (Painter, Mayer, & Cullinane, 1991).

First Steps to CALS Compliance published in the CALS Journal (Painter & Mayer, 1992).

Intelligent Support for Simulation Model Design from IDEF3 Descriptions presented at the IDEF Users Group Conference and published in the proceedings of that conference (Benjamin, Mayer, & Blinn, 1992).

Representation, Information Flow, and Model Integration published in (Charles J. Petrie, Ed.) Enterprise Model Integration: Proceedings of the First International Conference (Menzel, Mayer, & Sanders 1992).

An Integrated Approach to Cost Benefit Analysis presented at the IDEF Users Group Conference and published in the proceedings of that conference (Benjamin, Mayer, & Graul, 1993).

The Role of Ontology in Enterprise Integration presented at the IDEF Users Group Conference and published in the proceedings of that conference (Mayer, Menzel, Painter, & Benjamin, 1993).

Intelligent Support for Simulation Modeling: A Description-Driven Approach presented at the Summer Computer Simulation Conference and published in the proceedings of that conference (Benjamin, Mayer, & Blinn, 1993).

An Integrated Concurrent Engineering Environment for Life Cycle Management published in the Journal of Systems Integration (Blinn, Ackley, & Mayer, 1994).

The Role of IDEF0 and IDEF3 in Business Process Improvement presented at the IDEF Users Group Conference and published in the proceedings of that conference (Benjamin, Cullinane, Mayer, & Menzel, 1994).

A Framework and a Suite of Methods for Business Process Reengineering (Mayer, Benjamin, Caraway, & Painter, 1995a) in B. Kettinger & V. Grover (Eds.), Business Process Reengineering: A Managerial Perspective.

Towards a Method for Acquiring CIM Ontologies selected for publication in the *International Journal of Computer Integrated Manufacturing* (Benjamin, Menzel, & Mayer, 1995).

IDEF4 Object Oriented Design Method presented at the Society for Enterprise Engineering Conference and published in the proceedings of that conference (Keen and Schafrik, 1995).

The following papers were submitted to conferences and journals but were not selected for presentation or publication. These papers are available for distribution.

Wholistic Design, Engineering, and Manufacturing submitted to *IEEE Transactions on Engineering Management.* (Mayer, 1992c).

Theoretical Foundations for a Global Representational Medium submitted to the *International Journal of Production Research* (Menzel & Mayer, 1992).

An Ontology and Process Description Method for Design and Implementation of Information Integrated Enterprises submitted to the *International Journal of Flexible Automation & Integrated Manufacturing* (Mayer, 1992a).

Representing Knowledge for Concurrent Engineering submitted to *IEEE Special Issue on Concurrent Engineering* (Sanders, 1992).

Framework of Frameworks submitted to the *Journal of Systems Integration* (Mayer, 1992b).

IDEF∅ Function Modeling Method Formalization contributed to NIST IDEF∅ standardization activity and published as an internal NIST Technical Report (Menzel & Fulton, 1992).

Cognitive Skills in Simulation Modeling: The Role of Qualitative Reasoning being reviewed for publication in *IIE Transactions* (Mayer & Benjamin, 1993).

Technology Transfer Summary

This section has described the results of the technology transfer activity. The standards participation and documentation activity, coupled with IICE Applications work at OC-ALC, have made major strides in the dissemination and promotion of the IICE technology. The success of the technology transfer efforts can be measured by the influence of IICE technology on other projects and programs, and by the presence of IICE technology in the commercial sector.

REFERENCES

American National Standards Institute [ANSI]. (1975). *ANSI/X3/SPARC study group on database management systems,* (Interim Report 75-02-08, 7514TS01). *Association of Computing Machinery,* 7(2).

Anupindi, S. R. (1992). *Semantic requirements for an integrated bill of materials system.* Unpublished master's thesis, Texas A&M University, College Station, TX.

Banerjee, J., Chou, H., Garza, J., & Kim, W. (1986). *Data model issues for object-oriented applications* (MCC Techical Report Number: DB-099-86, rev. 1). Austin, TX: Microelectronics & Computer Technology Corporation.

Barwise, J. (1989). *The situation in logic.* Stanford University: Stanford University CSLI Publications.

Barwise, J., Gawon, J. M., Plotkin, G., & Tutiya, S. (Eds.). (1991). *Situation theory and its applications* (Vol. 2). CSLI Lecture Notes 26.

Barwise, J., & Perry, J. (1983). *Situations and attitudes.* Cambridge, MA: MIT Press.

Benjamin, P., Mayer, R. J., & Blinn, T. (1992, October). Intelligent support for simulation model design from IDEF3 descriptions. In R. Preston (Ed.), *IDEF Users Group Conference Proceedings,* np. Washington, D.C.: IDEF Users Group.

Benjamin, P., Mayer, R. J., & Graul, M. (1993, May). An integrated approach to cost benefit analysis. In R. Preston (Ed.), *IDEF Users Group Conference Proceedings,* 174-180. College Park, MD: IDEF Users Group.

Benjamin, P., Fillion, F., Mayer, R. J., & Blinn, T. (1993, July). Intelligent support for simulation modeling: A description-driven approach. In Schoen, J. (Ed.) *The Proceedings of the 1993 Summer Computer Simulation Conference,* 273-277.. San Diego, CA: The Society of Computer Simulation.

Benjamin, P., Cullinane, Thomas P., Mayer, Richard J., Menzel, Christopher P. (1994, May). The role of IDEFØ and IDEF3 in business process improvement. In *IDEF Users Group Conference Proceedings,* 158-170.. Richmond, VA: IDEF Users Group.

Benjamin, P., Menzel, C., & Mayer, R. J. (in press). Towards a method for acquiring CIM ontologies. *International Journal of CIM.*

Bertain, L. (Ed.). (1991). *CIM implementation guide.* Dearborn, MI: Society of Manufacturing Engineers.

Blinn, T., Mayer, R. J., & Joshi, S. (1991, May). Automated design of factory simulation models from IDEF3 process flow descriptions. In R. Preston (Ed.), *IDEF Users Group Conference Proceedings* (pp. 419-435). Albuquerque,. NM: IDEF Users Group.

Blinn, T., Ackley, K., & Mayer, R. J. (1994). An integrated concurrent engineering environment for life cycle management. *Journal of Systems Integration*, 2(4), 51-65.

Booch, G. (1991). *Object-oriented design with applications*. Redwood City, CA: The Benjamin/Cummings Publishing Company.

Brachman, R. J. (1983). What IS-A is and isn't: An analysis of taxonomic links in semantic networks. *Computer* 16(10), 30-36.

Burkhart, R., Dickson, S., Hanna, J., Perez, S., Sarris, T., Singh, M., Sowa, J., & Sundberg, C. (1991). *IRDS conceptual schema working paper* (ISO/IEC JTC1/SC21/WG3 N). Moline, IL. ANSI X3H3 Working Committee.

CAD Framework Initiative, Inc. (1991). *CAD framework initiative (CFI) tool encapsulation specification* (51, version. 0.22), *Design representation information model and programming interface* (121, version 0.7), *ntertool communication message dictionary* (ITC-90-G-04, version 0.6), *CAD framework initiative (CFI), inter-tool communication procedural interfaces* (ITC-90-G-02, version 0.19). Austin, TX: Microelectronics & Computer Technology Corporation.

CALS/CE Industry Steering Group. (1991). *A framework for concurrent engineering*. Washington,D.C. CALS/CE Industry Steering Group, Concurrent Engineering Framework Task Group.

Cannata, Phil. (1991, September). Carnot participation in demo brings interoperability closer. *Collaborations* 3(3). Austin, TX: Microelectronics & Computer Technology Corporation.

Chaitin, G. (1987). *Algorithmic information theory*. Cambridge: Cambridge University Press.

Coad, P., & Yourdon, E. (1991). *Object-oriented design*. Englewood Cliffs, NJ: Yourdon Press.

Coleman, Derek, Arnold, Patrick, Bodoff, Stephanie, Dollin, Chris, Gilchrist, Helena, Hayes, Fiona, & Jeremaes, Paul. (1994) *Object-oriented development: The fusion method*. Englewood Cliffs, NJ: Prentice Hall.

Control Data Corporation (CDC). (1982). *The reference and idea language (RIDL) conceptual language*. Arden Hills, MN: Control Data Corporation.

Clocksin, W. F. & Mellish, C. (1984). *Programming in PROLOG*. New York: Springer-Verlag.

Cullinane, T., Mayer, R. J., & McCollum, N. (1991, October). Application of IDEF3: A process flow modeling method. In R. Preston (Ed.), *IDEF Users Group Conference Proceedings*, 164-182. Kettering, OH: IDEF Users Group.

Datalogics, Inc. (1988). *Final report content data model of organizational level maintenance information for automated interchange of technical source data.* Chicago, IL: US Air Force Human Resource Laboratory, DataLogics, Inc.

Date, C. J. (1990). *An introduction to database systems* (5th ed.). *Vol. I: The systems programming Ser. II.* Reading, MA: Addison-Wesley.

Decker, L. P. and Mayer, R. J. (1992, Septermber). *Information System Constraint Language (ISyCL) Technical Report*, AL-TP-1992-0019, Armstrong Laboratory.

deWitte, P. S., Mayer, R., Su, C. J., Benjamin, P., Keen, A., Browne, D., Sun, T., Hari, U., Padmanaban, N., Wysk, R. (1992). *A knowledge-based process planning system (KAPPS) with assumption-based truth maintenance system and geometric reasoning system* (KBSI-KAPPS-91-SR-01-0492-01). Washington, D.C.: Defense Advanced Research Projects Agency (DARPA), Small Business Innovation Research Program.

Devlin, K. (1991). *Logic and information. Volume I: Situation theory.* Cambridge, MA: Cambridge University Press.

Drago, S. L., Technical point of contact. (1990). *Enterprise integration program (EIP) request for proposal.* Wright-Patterson AFB, OH: United States Air Force Systems Command, Aeronautical Systems Division.

Earl, M. (1989). *The content data model: A specification for integrated data bases of maintenance information.* Wright-Pattersoin AFB, OH: AFHRL/LRC.

Earl, M., & Gunning, D. (1990). *Content data model: Technical summary.* Wright-Patterson AFB, OH: Logistics and Human Factors Division.

Fillion, F. (1995). *Towards a formal language for the support of model integration* (Internal technical report). College Station, TX: Knowledge Based Systems, Inc.

Fulton, J. (1992). Enterprise integration using the semantic unification meta-model: In Petrie, Charles J. (Ed.) *Enterprise integration modeling: Proceedings of the first international conference*, 278-299. Cambridge, MA: MIT Press.

Genesereth, M. R., & Fikes, R. E. (1992). *Knowledge interchange format version 3.0 - reference manual* (Report Logic-92-1). Stanford, CA: Stanford University, Logic Group.

Goldfine, A., & Konig, P. (1987). *A technology overview of the information resource dictionary system* (rev. 1). Gaithersburg, MD: Center for Programming Science and Technology, Institute for Computer Science and Technology, National Bureau of Standards.

Goldratt, E. M. (1990). *Theory of constraints.* Croton-On-Hudson, NY: North River Press.

Gross, M., Ervin, S., Anderson, J., & Fleisher, A. (1987). Designing with constraints. In Y. E. Kaley (Ed.), *Computability of design.* New York: Wiley.

Gruber, T. R. (1992). *Ontolingua: A mechanism to support portable ontologies (KSL 91-66).* Stanford, CA: Stanford University.

Gruber, T. R. (1993). A translation approach to portable ontologies. *Knowledge Acquisition, 5*(2), 199-220.

Hobbs, J. R., Croft, W., Davies, T., Edwards, D., & Laws, K. (1987). *The TACITUS commonsense knowledge base.* Palo Alto, CA: Artificial Intelligence Research Center, SRI International.

Hughes, G., & Cresswell, M. (1968). *An introduction to modal logic.* London: Methuen.

Huhns, M. N. (1992). Enterprise information modeling and model integration in Carnot. In C. Petrie (Ed.), *Enterprise Integration Technology: Proceedings from the First International Conference.* Cambridge, MA: MIT Press.

IDEF Users Group, Working Group 1 (Frameworks) Technical & Test Committee. (1992). *IDEF Users Group Framework Document* (version 1.5). Kettering, OH: IDEF Users Group.

International Standards Organization. (1982). *Concepts and terminology for the conceptual schema and the information base* (ISO/TC97/SC5/WG3). Washington, D.C.: International Standards Organization.

International Standards Organization. (1987). *Information processing systems: Concepts and terminology for the conceptual schema and the information base* (ISO/TR 9007). Washington, D.C.: International Standards Organization.

International Standards Organization. (1988). *Framework for CIM system integration* (ISO T484/SC5/WG1-N84). Washington, D.C.: International Standards Organization.

Jacobsen, I. (1994). *Object-oriented software engineering: A use case driven approach.* Reading, MA: Addison-Wesley.

Jorgenson, B. R., & Walter, P. A. (1991). *Business entity relationship model (BERM)*. Seattle, WA: Boeing Computer Services-Commercial Airplane Support Systems Design Integration.

Jorgenson, B. R., & Walter, P. A. (1992). *Business relationship classification: Classes of relationships used in a business entity relationship model* (G-2675-TR92-004). Seattle, WA: Boeing Computer Services-Commercial Airplane Support Systems Design Integration.

Judson, D. L. (1985). *Integrated information support system (IISS): an evolutionary approach to integration, manufacturing technology division*. Wright-Patterson AFB, OH: Materials Laboratory, Air Force Wright Aeronautical Laboratories.

Keen, A. & Schafrik, F. (1995). *IDEF4 object oriented design method*. Presented at the Society for Enterprise Engineering Conference, Washington, D.C.

Knowledge Based Systems, Inc. (1991). *Detailed research plan, information integration for concurrent engineering program*. College Station, TX: Knowledge Based Systems, Inc.

Kuziak, A., & Dagli, C. (Eds.). (1994). *Intelligent systems in design and manufacturing*. Fairfield, NJ: ASME Press.

Lenat, D., Prakash, M., & Shepherd, M. (1986). CYC: Using common sense knowledge to overcome brittleness and knowledge acquisition bottlenecks. *The AI Magazine,* 7(2), pp 65-85.

Linn, J. L., & Winner, R. I. (1986). *The Department of Defense requirements for engineering information systems. Volume 1: Operational Concepts*. Alexandria, VA: EIS Requirements Team, The Institute for Defense Analyses.

Linn, J. L., & Winner, R. I. (1989). *The Department of Defense requirements for engineering information systems. Volume 2: Requirements*. Alexandria, VA: EIS Requirements Team, The Institute for Defense Analyses.

Martin, J., & Odell, J. (1992). *Object-oriented analysis and design*. Englewood Cliffs, NJ: Prentice Hall.

Mayer, R. J. (1991, November). A roadmap for enterprise integration. In A. Skomra (Ed.), *AutoFact '91 Conference Proceedings*, 7.1-7.26. Dearborn, MI: Society of Manufacturing Engineers.

Mayer, R. J., & Wells, M. S. (1991). *Integrated development support environment (IDSE) concepts and standards*. Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J. (1992a). *An Ontology and Process Description Method for Design and Implementation of Information Integrated Enterprises*. Manuscript submitted for publication.

Mayer, R. J. (1992b). Framework of frameworks (internal paper). College Station, TX: Knowledge Based Systems, Inc.

Mayer, R. J. (1992c). *Wholistic Design, Engineering, and Manufacturing.* Manuscript submitted for publication.

Mayer, R. J., Menzel, C., Painter, M. K., deWitte, P. S., Blinn, T. & Benjamin, P. (1992a). *IDEF3 process description capture method report.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Menzel, C., Painter, M. K., deWitte, P. S., Blinn, T. & Benjamin, P. (1992). *IDEF4 object-oriented design method report.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., & Benjamin, P. (1993). Cognitive skills in simulation modeling: The role of qualitative reasoning (internal paper). College Station, TX: Knowledge Based Systems, Inc.

Mayer, R. J., Menzel, C., Painter, M. K., & Benjamin, P. (1993). The role of ontology in enterprise integration. In R. Preston (Ed.), *Proceedings of the May 1993 IDEF Users Group Conference,* (pp. 162-168). College Park, MD: IDEF Users Group.

Mayer, R. J., Benjamin, P., Menzel, C., Fillion, F., deWitte, P. S., Futrell, M. T., & Lingineni, M. (1994). *IDEF5 ontology capture method report.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Benjamin, P., Caraway, B. E., & Painter, M. K. (1995a). A framework and a suite of methods for business process reengineering. In B. Kettinger & V. Grover (Eds.), *Business process change: Reengineering concepts, methods and technologies,* 245-290. Harrisburg, SC: Idea Group Publishing.

Mayer, R. J., Keen, A. A., Browne, D. C., Harrington, S., Marshall, C., Painter, M. K., Schafrik, F., Huang, J. C., Wells, M. S., & Hisesh, H. (1995b, in press). *IDEF4 /C++ object-oriented design method report.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Crump, J. W. IV, Fernandez, R., Keen, A., & Painter, M. K. (1995c, in press). *IDEF compendium.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Menzel, C., Painter, M. K., deWitte, P. S., Blinn, T. & Benjamin, P. (1995d, pending). *IDEF3 process description capture method report.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Menzel, C., Painter, M. K., deWitte, P. S., Blinn, T. & Benjamin, P. (1995e, pending). *IDEF4 object-oriented design method report.* Wright-Patterson AFB, OH: AL/HRGA.

Menzel, C. (1991, October). The importance of mathematical formalization for advancing modeling methods technology. In R. Preston (Ed.), *IDEF Users Group Conference Proceedings*, 83-102. Fort Worth, TX: IDEF Users Group.

Menzel, C., & Mayer, R. J. (1992). Theoretical foundations for a global representational medium (internal paper). College Station, TX: Knowledge Based Systems, Inc.

Menzel, C., Mayer, R. J., & Sanders, L. (1992). Representation, information flow, and model integration. In C. Petrie (ed.), *Proceedings of the International Conference on Engineering Integration and Modeling Technology*, Cambridge, MA: MIT Press.

Neches, R., Fikes, R., Finin, T., Gruber, T., Patil, R., Senator, T., & Swartout, W. R. (1991). Enabling technology for knowledge sharing. *AI Magazine, 12*(3), 36-56.

Novak, J., & Gowin, D. B. (1984). *Learning how to learn.* Cambridge, MA: Cambridge University Press.

Painter, M. K. (1991, May). IICE: program foundations and philosophy. In R. Preston (Ed.), *IDEF Users Group Conference Proceedings*, 186-199. Fort Worth, TX: IDEF Users Group.

Painter, M. K. (1991). CIM: modeling with and IDEF perspective. In L. Bertain (Ed), *CIM Implementation Guide.* Dearborn, MI: Society of Manufacturing Engineers.

Painter, M. K., Mayer, R. J., & Cullinane, T. (1991, November). The many faces of concurrent engineering. In A. Skomra (Ed.), *AutoFact '91 Conference Proceedings*, 17.1-17.25. Dearborn, MI: Society of Manufacturing Engineers.

Painter, M. K., & Mayer, R. J. (1992). First steps to CALS compliance. *CALS Journal, 1*(1), 27-30.

Painter, M. K., Graul, M., & Marshall, C. (1995, in press). *IICE Technology Transition Effort for E-3 Programmed Depot Maintenance (PDM) Final Report.* Wright-Patterson AFB, OH: AL/HRGA.

Petrie, C. J. (Ed.). (1992). *Enterprise integration modeling: Proceedings of the first international conference.* Cambridge, MA: MIT Press.

Ramey, T. L. (1983). *Guidebook to system development.* Los Angeles, CA: Hughes Aircraft Company.

Reddy, Y. V. (1989). *DARPA initiative in concurrent engineering (DICE): red book of functional specifications ffor the DICE architecture* (MDA972-88-C-0047). Morgantown, West VA: West Virginia University, Concurrent Engineering Research Center.

Rockwell International. (1989). *Integrated design support system (IDS)* (AFHRL-TR-89-6). Wright-Patterson AFB, OH: AFHRL.

Rumbaugh, J. (1991) *Object-oriented modeling and design.* Englewood Cliffs, NJ: Prentice-Hall.

Sanders, L., Mayer, R. J., Browne, D. C., & Menzel, C. (1991, November). Container objects: a description based knowledge representation scheme. In A. Skomra (Ed.), *AutoFact '91 Conference Proceedings,* 7.39-7.50. Dearborn, MI: Society of Manufacturing Engineers.

Sanders, L. (1992). Representing knowledge for concurrent engineering (internal paper). College Station, TX: Knowledge Based Systems, Inc.

Sarris, A. K. (1992). Needs analysis and requirements document: integration toolkit and methods, corporate data integration tools (*MANTECH Report,* WL-TR-92-8027). Wright-Patterson AFB, OH.

Shannon, C., & Weaver, W. (1949). The mathematical theory of communication. Urbana: University of Illinois Press.

Shlaer, S., & Mellor, S. (1988). *Object-oriented systems analysis: Modeling the real world in data.* Englewood Cliffs, NJ: Prentice Hall.

Shlaer, S., & Mellor, S. (1992). *Object lifecycles: Modeling the world in states.* Englewood Cliffs, NJ: Yourdon Press.

Smith, G. F., & Browne, G. J. (1993). Conceptual foundations of design problem solving. *IEEE transactions on systems, man, and cybernetics. 23*(5), 1209-1219.

Sowa, J. (1984). *Conceptual structures: Information processing in mind and machine.* Reading, MA: Addison-Wesley.

Thayse A., (Ed.). (1991). *From modal logic to deductive databases.* New York: John Wiley & Sons.

van Griethugen, J. J. (1982). *Concepts and terminology for the conceptual schema and the information base* (ISO/TC97/SC5). 5600 MD Eindhoven, The Netherlands: International Organization for Standardization.

Wirths-Brock R., Wilkerson, B., & Wiener, L. (1990) *Designing object-oriented software.* Englewood Cliffs, NJ: Prentice Hall.

Zachman, J. (1986). A framework for information systems architecture. *IBM Systems Journal, 26*(3), 276-292.

BIBLIOGRAPHY

Aczel, P., Israel, D., Katagiri, Y., & Peters, S. (Eds.). (1993). *Situation theory and its applications* (Vol. 3). CSLI Lecture Notes 37.

Allen, J. F. (1984). Towards a general theory of action and time. *Artificial Intelligence, 23,* 123-154.

Balci, O., & Nance, R. E. (1987). Simulation model development environments: A research prototype. *Journal of the Operational Research Society, 38*(8), 753-763.

Bealer, G. (1980). *Quality and concept.* Oxford: Oxford University Press.

Chaffin, R., & Herrmann, D. J. (1987). Relation element theory: A new account of the representation and processing of semantic relations. In D. S. Gorfein (Ed.), *Memory and Learning: The Ebbinghaus Centennial Conference.* Hillsdale, NJ: Erlbaum.

Chen, P. (1976). The entity-relationship model: Toward a unified view of data. *ACM Transactions on Database Systems, 1*(1), 9-36.

Cohen, N. J. (1987). Preserved learning capacity in amnesia: evidence for multiple memory systems. In X. Editor (Ed.), *Neuropsychology of memory* (pp. 83-103). New York: Guilford Press.

Coleman, D. S. (1989). *A framework for characterizing the methods and tools of an integrated system engineering methodology (ISEM)* (draft 2, rev. 0). Santa Monica, CA: Pacific Information Management, Inc.

Coleman, S. (1991). *Information engineering practioner's guide: Volume IV — Business area analysis.* Culver City, CA: Pacific Information Management, Inc.

Cross, S. (1983). *Qualitative reasoning in an expert system framework* (T-124). Urbana, IL: University of Illinois Coordinated Science Lab.

De Kleer, J. (1979). *Causal and teleological reasoning in circuit recognition* (TR-529). Cambridge, MA: MIT AI Lab.

De Kleer, J., & Brown, J. S. (1984). A qualitative physics based on confluences. *Artificial Intelligence, 24,* 7-83.

Enderton, H. (1972). *A mathematical introduction to logic.* New York: Academic Press.

Forbus, K. (1984). Qualitative process theory. *Artificial Intelligence, 24,* 85-168.

Fox, M. (1992). The TOVE project: Towards a commonsense model of the enterprise. In C. Petrie (Ed.), *Enterprise Integration Technology: Proceedings from the First International Conference* (pp. 310-319). Cambridge, MA: MIT Press.

Futrell, M. T. (1991). *The IDEF5 application procedure.* Unpublished master's project report, Texas A&M University, College Station, TX.

Gabriel, R. P., White, J. L., & Bobrow, D. G. (1991). Integrating object-oriented and functional programming. *Communications of the ACM, 34*(9), 29-38.

Goldratt, E. M., & Cox, J. (1986). *The goal* Croton-On-Hudson, NY: North River Press.

Goldratt, E. M., & Fox, Robert E. (1986). *The race.* Croton-On-Hudson, NY: North River Press.

Goldratt, E. M. (1991). *The haystack syndrome.* Croton-On-Hudson, NY: North River Press.

Goldratt, E. M. (1992, July). *An introduction to theory of constraints: The production approach.* Two-day workshop course materials.

Gruber, T. R. (1993). *Towards a knowledge medium for collaborative product development.* Workshop Notes, AAAI-92 Workshop Program A in Enterprise Integration.

Guha, R. V., & Lenat, D. V. (1990). Cyc: A mid-term report. *AI Magazine, 11*(3), 32-59.

Gunning, D. (1988). *A general framework for describing human-gumputer information systems.* Wright-Patterson AFB, OH: AFHRL.

Herbert, R. (1992). *Issues: Three-schema architecture.* Long Beach, CA: Douglas Aircraft Company.

Herbert, R., & Krasowski, M. (1992). *Information systems integration: Summary of approaches at douglas aircraft company.* Long Beach, CA: Douglas Aircraft Company.

IDEF Users Group. (1990). *IDEF: Framework* (draft report ; IDEF-U.S.-0001). Dayton, OH: IDEF Users Group.

Jackson, P. (1990). *Introduction to expert systems.* Reading, MA: Addison-Wesley.

Keen, A. A., & Mayer, R. J. (1991, November). *Approaches to design object management.* Paper presented at the AutoFact 1991 meeting of the Society of Manufacturing Engineers, Chicago, IL.

Knowledge Based Systems, Inc. (1990). *A design knowledge management system (DKMS)* (SBIR Phase I Final Report). Wright-Patterson AFB, OH: AFHRL.

Knowledge Based Systems, Inc. (1991). *Formal foundations for an ontology description method* (KBSI-SBONT-91-TR-01-1291-02). College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1991). *Knowledge based information model integration.* College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1991). *The nature of ontological knowledge: A manufacturing systems perspective* (KBSI-SBONT-91-TR-01-129-01). College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1991). *Ontology acquisition method requirements document* (KBSI-SBONT-91-TR-01-1291-03). College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1991). *Ontology capture tool requirements document* (KBSI-SBONT-91-TR-01-1291-04). College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1991). *Reliable object-based architecture for intelligent controllers.* College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1992). *IDEF4 method report.* College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1992). *Ontology capture tool: Object-oriented design document* (KBSI-SBONT-91-TR-0292-01). College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1992). *Situation-based ontology: Phase I report.* College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1993). *Ontology-driven information integration: Phase I final report.* College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems, Inc. (1994). *IDEF5 method report.* College Station, TX: Knowledge Based Systems, Inc.

Knowledge Based Systems Laboratory. (1991). *IDEF4 technical report* (KBSL-89-1004). College Station, TX: Texas A&M University.

Kosanke K. (1992). CIMOSA - A European development for enterprise integration. Part 1: An overview. In C. Petrie (Ed.), *Enterprise integration technology: Proceedings from the First International Conference* (pp. 179-188). Cambridge, MA: MIT Press.

Kripke, S. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica, 16,* 39-48.

Lin, M. J. (1990). *Automatic simulation model design from a situation theory based manufacturing system description.* Unpublished doctoral dissertation, Texas A & M University, College Station, TX.

Link, G. (1983). The logical analysis of plurals and mass terms: A lattice theoretic approach. In R. Bauerle (Ed.), *Meaning, use, and interpretation.* Berlin: De Gruyter.

Link, W. R., Von Holle, J. C., & Mason, D. (1987, November). *Integrated maintenance information system (IMIS): A maintenance information delivery concept* (AFHRL-TP-87-27). Brooks AFB, TX: Air Force System Command.

Loiselle, C. L., & Cohen, P. L. (1989). Explorations in the contributors to plausibility (COINS Technical Report 89-29). Amherst, MA: University of Massachusetts.

Maher, M. L. (1989). Synthesis and evaluation of preliminary designs. In J. S. Gero (Ed.), *Artificial intelligence in design.* New York: Springer-Verlag.

Martin, J. (1982). *An information systems manifesto.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Mayer, R. J. (1988). *Cognitive skills in modeling and simulation.* Unpublished doctoral dissertation, Texas A&M University, College Station, TX.

Mayer, R. J. (1990). *A framework generator* (draft technical report). College Station, TX: Knowledge Based Systems Laboratory.

Mayer, R. J. (Ed.). (1990). *IDEFØ function modeling: A reconstruction of the original Air Force report.* College Station, TX: Knowledge Based Systems, Inc.

Mayer, R. J. (Ed.). (1990). *IDEF1 information modeling: A reconstruction of the original Air Force report.* College Station, TX: Knowledge Based Systems, Inc.

Mayer, R. J. (Ed.). (1990). *IDEF1X data modeling: A reconstruction of the original Air Force report.* College Station, TX: Knowledge Based Systems, Inc.

Mayer, R. J. (1991). *Framework foundations research report* (Final Report). Wright-PattersonAFB, OH: AFHRL/LRA

Mayer, R. J., & deWitte, P. S. (1991). *Framework research report.* Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., deWitte, P. S., Griffith, P., & Menzel, C. (1991). *IDEF6 concept report*. Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Edwards, D. A., Decker, L. P., & Ackley, K. A. (1991). *IDEF4 technical report*. Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Menzel, C. P., & deWitte, P. S. (1991). *IDEF3 technical report*. Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., Menzel, C. P, & Mayer, P. S. (1991). *IDEF3: A methodology for process description*. Wright-Patterson AFB, OH: AL/HRGA.

Mayer, R. J., & Painter, M. K. (1991). *IDEF family of methods*. College Station, TX: Knowledge Based Systems, Inc.

Mayer, R. J. (1992). Enabling concurrent engineering through the auto-generation of geometric models for analysis from product specification data (internal paper). College Station, TX: Knowledge Based Systems, Inc.

McGraw, K., & Briggs, K. (1989). *Knowledge acqusition principles and guidelines*. New York: Prentice-Hall.

Menzel, C. (1990). Actualism, ontological commitment, and possible world semantics. *Synthese, 85*, 355-389.

Menzel, C. P., & Mayer, R. J. (1990). *IDEF3 formalization report*. Wright-Patterson AFB, OH: AL/HRGA.

Menzel, C. (1991). The true modal logic. *Journal of Philosophical Logic, 20*, 331-374.

Menzel, C., & Mayer, R. J. (1991). *Theoretical foundations for information representation and constraint specification* (AL-TP-1991-0044). Wright-Patterson AFB, OH: Human Resources Directorate, Logistics Research Division.

Menzel, C. P., Mayer, R. J., & Edwards, D. E. (1991). IDEF3 process descriptions and their semantics. In A. Kuziak & C. Dagli (Eds.), *Knowledge base systems in design and manufacturing*. New York: Chapman Publishing.

Meyer, B. (1987). Reusability: The case for object -oriented design. *IEEE Software, 4*(2), 50-64.

Musen, M. A. (1989). Conceptual models of interactive knowledge-acquisition tools. *Knowledge Acquisition, 1*, 73-88.

Olle, T. W., Hagelstein, J., Macdonald, I. G., Rolland, C., Sol, H. G., Assche, F. J. M., & Verrijn-Stuart, A. A. (1988). *Information systems methodologies: A framework for understanding.* Reading, MA: Addison-Wesley Publishing Company.

Overstreet, C. M., & Nance, R. E. (1985). A specification language to assist in analysis of discrete event simulation models. *Communications of the ACM, 28*(2), 190-201.

Pegden, C. D. (1982). *Introduction to SIMAN.* State College, PA: Systems Modeling Corporation.

Poole, D. (1990). A methodology for using a default and abductive reasoning system. *International Journal of Intelligent Systems, 5,* 521-548.

Pressman, R. S. (1987). *Software engineering: A practitioner's approach.* New York: McGraw-Hill.

Pritsker, A. A. B., & Pegden, C. D. (1979). *Introduction to simulation and SLAM.* New York: Halsted Press.

Shlaer, S., & Mellor, S. (1992). *Object lifecycles: Modeling the world in states.* Englewood Cliffs, NJ: Yourdon Press.

Silver, G. A., & Silver, M. L. (1989). *Systems analysis and design.* Reading, MA: Addison-Wesley.

Soley, R. M. (Ed.). (1990). *Object management architecture guide.* Farmington, MA: Object Management Group, Inc.

Stroustrup, B. (1994). *The C++ programming language.* Reading, MA: Addison-Wesley.

Taylor, D. (1990). *Object-oriented technology: A manager's guide.* Reading, MA: Addison-Wesley.

Udell, J. (1994, May) Componentware. *Byte,* 46-56.

Wallace, R. H., Stockenberg, J. E., & Charette, R. N. (1987). *A Unified methodology for developing systems.* New York: McGraw-Hill.

Winston, M.E., Chaffin, R., & Herrmann, D. (1987). A Taxonomy of part-whole relations. *Cognitive Science, 11,* 417-444.

Wittgenstein, L. (1953). *Philosophical Investigations.* Oxford: Basil Blackwell.

Yourdon, E., & Constantine, L. L. (1979). *Structured design: fundamentals of a discipline of computer program and systems design.* Englewood Cliffs, NJ: Prentice-Hall.

Zoetekouw, D. (1992) An overview of formal semantics specification methods. In C. Petrie (Ed.), *Enterprise Integration Technology: Proceedings from the First International Conference* (pp. 216-225). Cambridge, MA: MIT Press.

ACRONYMS

IDL, 3

IDS, 173

IDSE, 2, 3, 27

IEC, 132

IEEE, 7, 8, 102

IGES, 132, 178

IICE, i, 6, 27

IMIS, 172

IPO, 132

IRDS, 119, 130

IRL, 99

ISCs, 58

ISM, 31

ISO, 103

IST, 28

ISyCL, 1, 24, 32

J

JTC1, 132

K

KBSI, 1, 7, 94

KIF, 130, 173

KR, 130

KSE, 127

L

LCA, 30

LCOM, 51, 169

M

MCC, 35

MIS, 68

MIT, 102

N

NASA, 95

NC, 126

NIRS, 1, 2, 8, 23

NIST, 3

NSF, 95

O

O&A, 70

OC-ALC, 8, 65

OCBT, 50, 51

OLE, 62

OQL, 130

P

PDES, 9, 102

PDM, 2, 65

PMI, 138

PROLOG, 109

R

RIDL, 172

ROSE, 170

RRL, 51

S

SBIR, 94

SC21, 132

SGML, 52, 172

SPARC, 159

SQL, 130

SRRP, 73

STEP, 132

SUMM, 3, 8

T

TO, 68

TSA, 157

U

USAF, 102

X

X3, 132

X3T2, 132

APPENDIX A
METHOD OVERVIEWS

IDEF3 Process Description Capture Method Overview

The IDEF3 Process Description Capture Method is used to collect and document information about processes. The IDEF3 method formalizes proven practices used to capture process knowledge and provides an expressively powerful language for information capture and expression. These two dimensions of IDEF3 — the procedure embodying proven practices and an expressively powerful language — work together to focus user attention on relevant aspects of a process and provide the necessary expressive power to explicitly represent information about the nature and structure of that process.

IDEF3 process descriptions can be used to perform the following tasks:

1.   Record the raw data resulting from fact-finding interviews.

2.   Determine the impact of an organization's information resources on the major processes of an enterprise.

3.   Document decision procedures affecting the states and life cycle of critical shared data.

4.   Manage data configuration and define change control policy.

5.   Support system design including design trade-off analysis.

6.   Support simulation model generation.

IDEF3 is used to capture the behavioral aspects of an existing or proposed system. Captured process knowledge is structured in the context of a *scenario*, making IDEF3 an intuitive knowledge acquisition device for describing a system. IDEF3 is used to capture temporal information, including precedence and causality relationships associated with enterprise processes. The resulting IDEF3 descriptions provide a structured knowledge base for constructing analytical and design models.

Unlike simulation languages that build predictive mathematical *models*, IDEF3 builds structured *descriptions*. These descriptions capture information about what a system actually does or will do, and provide for the organization and expression of different user views of the system.

IDEF3 users have cited benefits in terms of cost savings, schedule gains, quality improvements, organic capability improvements, and lasting changes to organizational culture.[29] These gains have been realized by using IDEF3 to:

1. Identify previously obscure process links between organizations.

2. Highlight redundant and/or non-value-added activities.

3. Design new processes.

4. Speed the efficiency and effectiveness of new worker training.

Additional benefits gained through the use of IDEF3 have been realized through its use as a mechanism to:

1. Raise awareness of external customer needs and the business-sustaining processes for servicing those needs.

2. Capture and distribute detailed manufacturing process knowledge among geographically dispersed organizational units.

3. Determine the impact of an organization's information resources on the major operating scenarios of an enterprise.

4. Provide an implementation-independent specification for efficient information delivery and human-system interaction.

5. Define data configuration management and change control policies.

6. Document the decision procedures affecting the states and life cycle of critical shared data (particularly manufacturing, engineering, maintenance, and product definition data).

7. Support system design and design tradeoff analysis.

8. Minimize the impact of organization personnel involvement in the development of high-quality IDEFØ function models.

9. Expedite the development and validation of simulation models.

_____

[29] See, for example, papers describing the use of IDEF3 in the Proceedings of the IDEF Users Group.

10. Develop real-time control software by providing a mechanism for clearly defining facts, decision points, and job classifications.

11. Define the behavior of workflow management systems and applications.

12. Prescribe the process by which evolutionary change within an organization or system will be achieved.

13. Collect the information needed to perform critical path analyses, resource leveling, and milestone setting activities for project management.

14. Establish a framework for, and facilitate, Activity Based Costing (ABC).

15. Develop templates of "world-class" business processes.

Using IDEF3 to Capture Process Descriptions. The description capture and validation process is generally undertaken to accomplish some purpose. That purpose may be simply to document a process — in which case the development of schematics and elaborations is an end unto itself. However, in most cases, description development is undertaken to assist with discovery or decision-making activities. Whether IDEF3 is used to document a process or to assist with discovery and decision-making, the same general steps are applied recursively.

1. Collect — Acquire observations and written descriptions of processes.

2. Classify — Individuate situation types, objects, object types, object states, and relations.

3. Organize — Assemble the data that have been collected and classified using the organizational structures provided in IDEF3.

4. Validate — Ensure the statements made in IDEF3 are grammatical, and that they corroborate with the collected descriptions of the situation.

5. Refine — Make adjustments to the existing structures to incorporate newly discovered information, to simplify the presentation, or to highlight important elements of interest.

Developing IDEF3 descriptions involves the creation of Process Schematics, Object Schematics, and their associated elaborations. Thus, two description modes, which focus attention on two perspectives of a process, are supported by the IDEF3 method. The first is a process-centered view supported by a Process Schematic. The second, an object-centered view of a process, is supported by an Object Schematic. A Process Schematic captures information about the way things work in an organization (e.g., a description of what happens to a part as it flows through a sequence of manufacturing processes). Object Schematics summarize how various kinds of objects are

199

transformed into other kinds of things, or how objects of a given kind change states through a process. Both the Process and Object Schematics contain units of information that make up the system description. These model entities form the basic units of an IDEF3 description.

The basic syntactic elements of the IDEF3 Schematic Language are shown in Figure A-1. The basic building blocks of IDEF3 process descriptions are the following.

1.   Unit of Behavior Symbols — Used to depict events, decisions, activities, processes, and so forth.

2.   Junctions — Used to depict branching or logic in a process execution or in object state change behavior.

3.   Links — Used to show logical or temporal relations between processes or between object states.

4.   Object Symbols — Used to depict objects in a given state.

5.   Referents — Used to highlight a participating object or a break in the normal flow of a process.

6.   Elaborations — Used to capture details about the objects involved in a process, their roles, the facts that hold during the process, and the constraints that govern the process.

7.   Notes — Used to provide additional information about a particular IDEF3 model element or to attach illustrations, text, screen layouts, comments, etc. to the description.

Each basic building block in the IDEF3 process description has its own elaboration and a textual description referred to as the "glossary."

Process Schematic Symbols	Object Schematic Symbols

Process Schematic Symbols

UOB Symbols

```
┌─────────────────────────┐
│                         │
│       UOB Labels        │
│                         │
├────────────┬────────────┤
│ Node Ref # │ IDEF Ref # │
└────────────┴────────────┘
```

Links

→ Precedence Links

→■→

→▶→

→◀→

→◆→

— — — — — Relational Links

Junctions

| & | – AND | | & | – Synchronous AND |

| O | – OR | | O | – Synchronous OR |

| X | – XOR |

Object Schematic Symbols

Object Symbols   ### Individual Symbols

(Object State Label)   (Individual Label ●)

Links

→ Weak Transition Link

→▶ Strong Transition Link

[Relation Label] n – Place First-order Relation Symbol

▶ Relation Label 2 – Place Second-order Relation Symbol

Junctions   ### Connecting Symbols

(&) – AND   ────→────

(O) – OR   ──────────

(X) – XOR   ○

Referents and Notes

Call and Continue Referent

```
┌──┬──────────────┐
│  │ Referent Type/│
│  │ Label         │
├──┴──────────────┤
│    Locator      │
└─────────────────┘
```

Call and Wait Referent

```
┌──┬──────────────┐
│  │ Referent Type/│
│  │ Label         │
├──┴──────────────┤
│    Locator      │
└─────────────────┘
```

Note

```
┌─────────────────┐
│ Note ID         │
│                 │
│               ╱ │
└──────────────╱  │
```
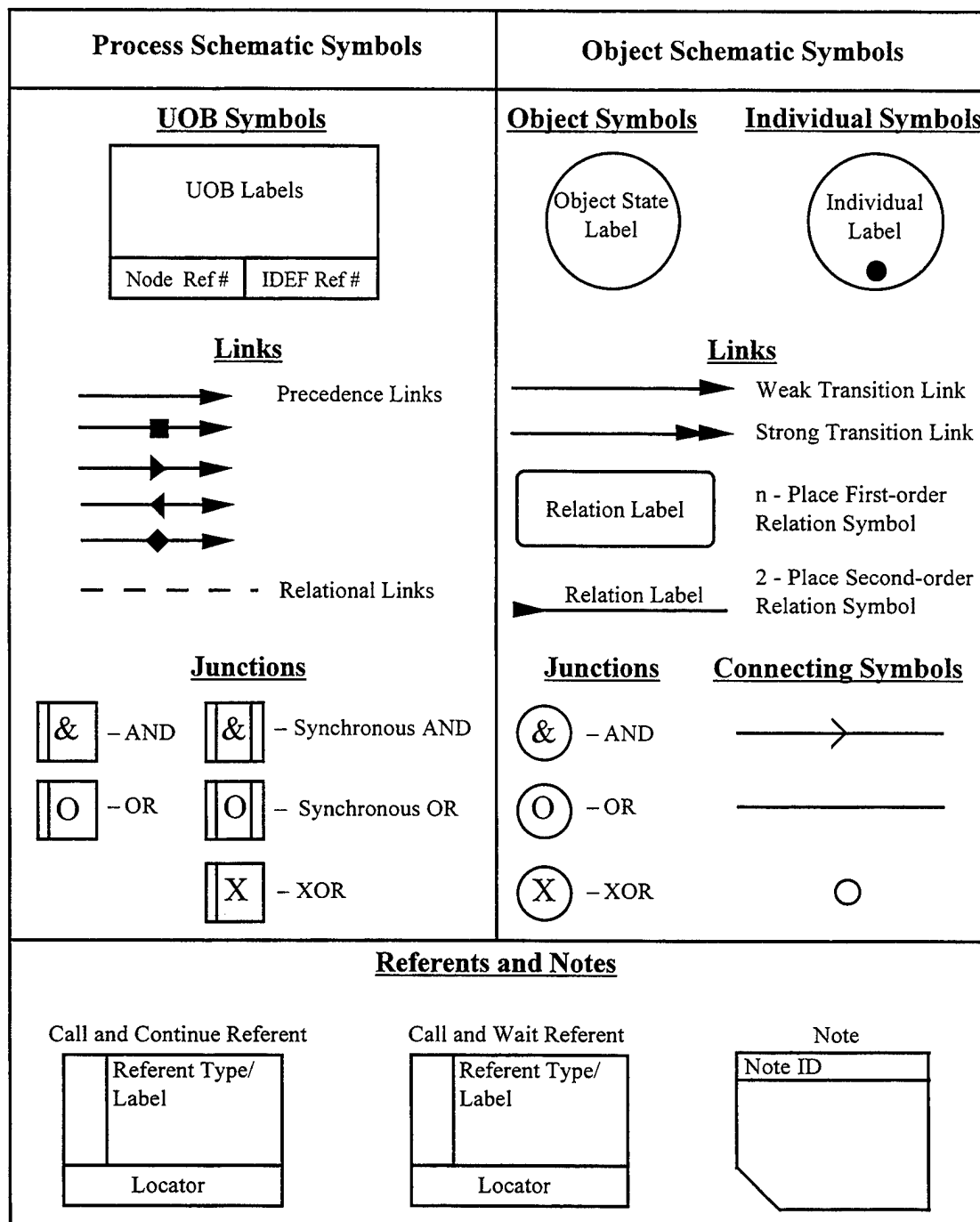
Figure A-1

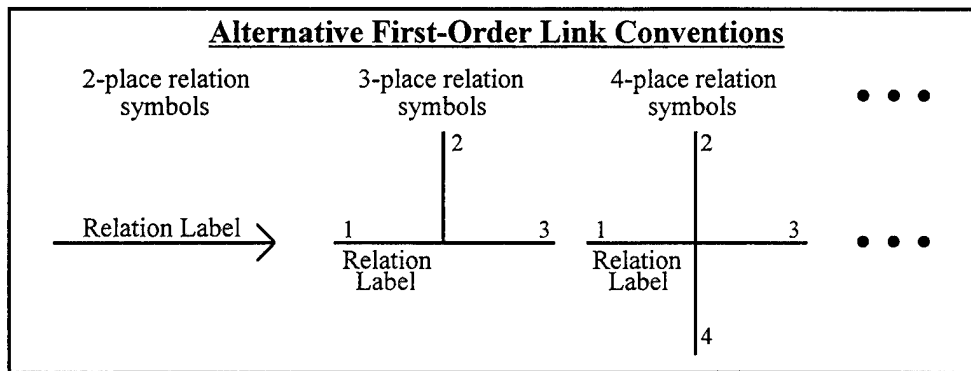**Symbols Used for IDEF3 Process Descriptions**

Figure A-2

Alternative Symbol Conventions for First-Order Links

The following example illustrates how the building blocks of the IDEF3 method can describe a typical scenario found in the business environment. The situation described is a Purchase Order process. A sample of the interview notes used to construct the graphical representation are as follow.

Interviewer: How does one order material?

Domain Expert: The first thing we do is request material using a Purchase Request form. Then the Purchasing department either identifies our current supplier for the kind of material requested or sets out to identify potential suppliers. If we have no current supplier for the needed item, Purchasing requests bids from potential suppliers and evaluates their bids to determine the best value. Once a supplier is chosen, Purchasing orders the requested material.

Interviewer: How does someone request material?

Domain Expert: Those requesting material must first prepare a Purchase Request. The requester must then obtain the Account Manager's approval, or that of the designated backup, for the purchase. Purchase Requests submitted for Account Manager approval must include the Account Number for the Project that will fund the purchase. Account Managers, or their designated backup, are responsible for, and must approve, all purchases made against their project accounts. After the Account Manager approves the purchase, an authorization signature may be required. To avoid a potential conflict of interest, the requester cannot be the same individual who approves or authorizes the request. Purchase Requests involving Direct projects require an authorization signature whereas Indirect projects do not. Once all the appropriate signatures are in place, the requester submits the signed Purchase Request to Purchasing. Purchasing then orders the requested material. The Purchase Request is thereafter tracked as an issued Purchase Order.

202

Figure A-3 provides a graphical representation of the scenario described by the domain expert. The activities described in the scenario appear as labeled boxes; these boxes can describe activities, processes, events, and so forth. The IDEF3 term for elements represented by these boxes is UOB. The arrows (links) tie the boxes (activities) together and define the logical flows. The smaller boxes define junctions that provide a mechanism for introducing the process logic.

Each UOB can have associated descriptions in terms of other UOBs called *decompositions* (Figure A-3) and in terms of a set of participating objects and their relations called an *elaboration*. A decomposition is a closer look at a particular UOB; that is, it is a more fine-grained IDEF3 representation of that UOB. The decomposition of *Request Material* in Figure A-3 is an example. Multiple views (decompositions) are supported in IDEF3. Variances among these decompositions may lead to the discovery of gaps in the domain expert's knowledge about certain elements of the process or highlight opportunities for process improvement.
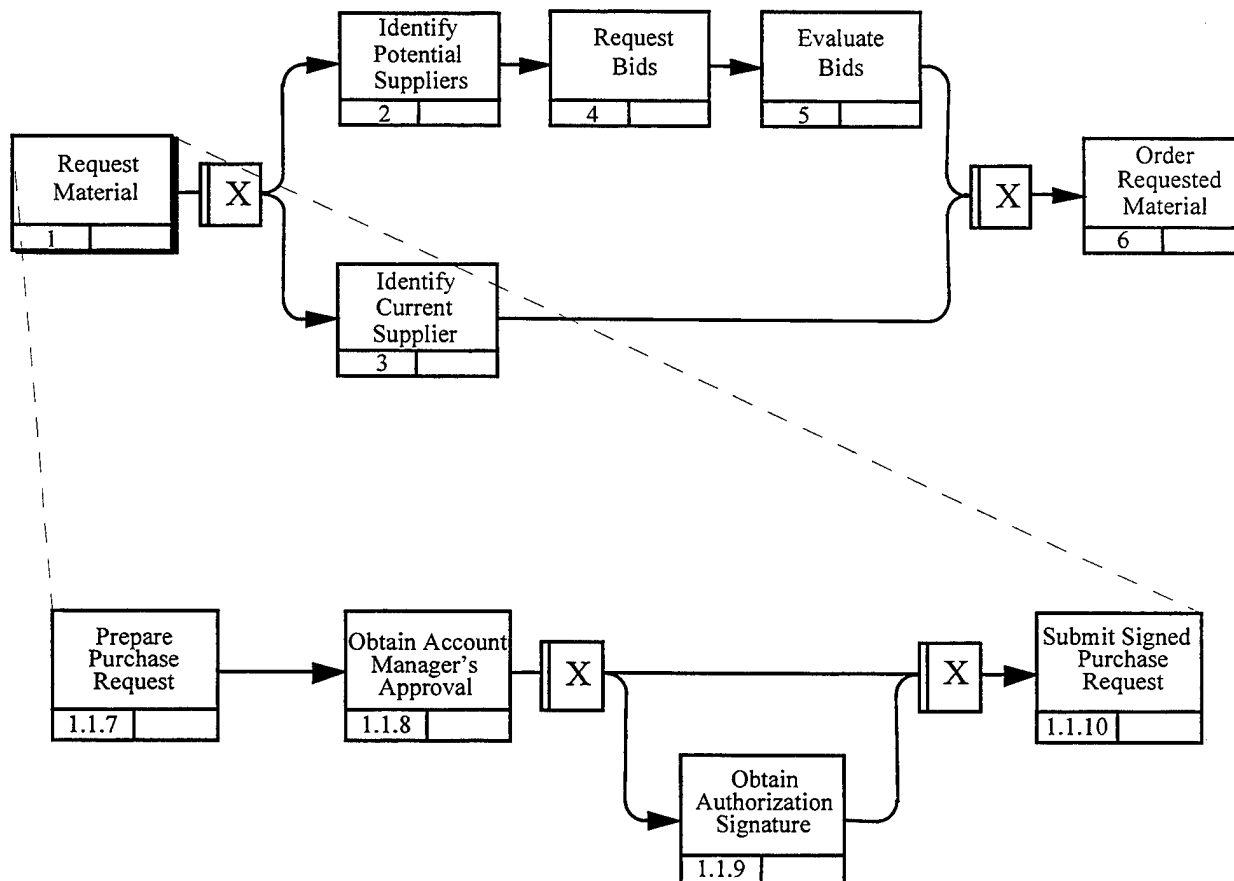

Figure A-3

Example IDEF3 Process Description

An elaboration captures the objects, constraint declarations, and other informative statements defined for instances of an activity. Each element of an IDEF3 description can have an elaboration where this information (e.g., resource requirements, roles played by objects in an instance of the process) is captured.

To provide a detailed characterization of the objects that participate in a process, it is useful to construct Object Schematics. Typically developed only for the important objects of the process description, Object Schematics provide a different view of the process being described (i.e., an object-centered view). Object Schematics are most often developed after the Process Schematic; however, some users find it easier to begin with the Object Schematic.

In the Purchase Order process example presented above, the Purchase Request may be the important object. If so, it may be useful to conceptualize the Purchase Request as an object in transition through several states in the process being described. This would involve examining the process from an object-centered view, beginning with the initial development of a Purchase Request through the eventual issuance of a Purchase Order. The passages below illustrate the process of developing an Object Schematic with this focus.

Having determined the main object of focus, the analyst begins Object Schematic development by identifying candidate object states. Because the key object of interest is the Purchase Request, it is necessary to examine the perceived changes in the Purchase Request's state through the process. The terminating state, as indicated by the client, is when the Purchase Request finally becomes a Purchase Order. Although many possible states of a Purchase Order are likely, the clients needs in this example do not require exploring those states. A number of sources either directly identify these states or provide clues that indicate the possibility that domain experts distinguish a state. These sources include raw descriptions provided by the domain expert, candidate UOB names, the objects themselves, and UOB elaboration forms. Identifying the possible state changes an object may undergo in a process often requires that the analyst work with domain experts to either extract or provide candidate names for object states.

By reviewing the data provided in the interviews, the analyst produces a list of candidate object states like the one below.

Purchase Request: Incomplete

Purchase Request: Completed

Purchase Request: Approved

Purchase Request: Authorized

Purchase Request: Submitted

Purchase Order: Issued

This list is likely to change as logically identical states are recognized, names are refined, and previously unnamed or unrecognized states are identified. The final Object Schematic depicted in Figure A-4 illustrates this point. For example, in addition to establishing that a Purchase Request might be *approved*, the final Object Schematic indicates that it might also be *disapproved*. As illustrated in this example, possible alternative paths of state change behavior are indicated by junction symbols in the Object Schematic. Additionally, the processes involved in each state transition can be indicated directly in the Object Schematic.[30]

Summary. IDEF3 help users capture and analyze the vital processes of an existing or proposed system. Procedural guidelines and simple-to-use graphical language structures aid users in capturing and organizing process information for multiple downstream uses. IDEF3's unique design includes the ability to capture and structure descriptions of how a system works from multiple viewpoints, enabling users to capture information conveyed by knowledgeable experts about the behavior of a system rather than directing user activity toward constructing engineering *models* to approximate system behavior. This feature is one of the central distinguishing characteristics separating IDEF3 from alternative process modeling methods. As an integral member of the IDEF family of methods, IDEF3 works well independently or in concert with other IDEF methods to identify and develop the vital processes of the business, ultimately optimizing the performance of those processes.

A more detailed description of IDEF3 can be found in the *IDEF3 Process Description Capture Method Report* ([Mayer et al, 1992] and [Mayer et al, forthcoming]).

IDEF4 Object-Oriented Design Method Overview

IDEF4 is an object-oriented design method for developing component-based client/server systems. It has been designed to support smooth transition from the application domain and requirements analysis models to the design and to actual source code generation. It specifies design objects with sufficient detail to enable source code generation. IDEF4 provides a bridge between domain analysis and implementation (Figure A-5).

---

[30] A summary-level description of the syntax and semantics used in the Object Schematic is provided below in the Object State Schematics section of the IDEF5 Ontology Description Capture Method summary. The synergy between process knowledge capture and ontology development motivated extensive reuse of schematic elements between these two methods.
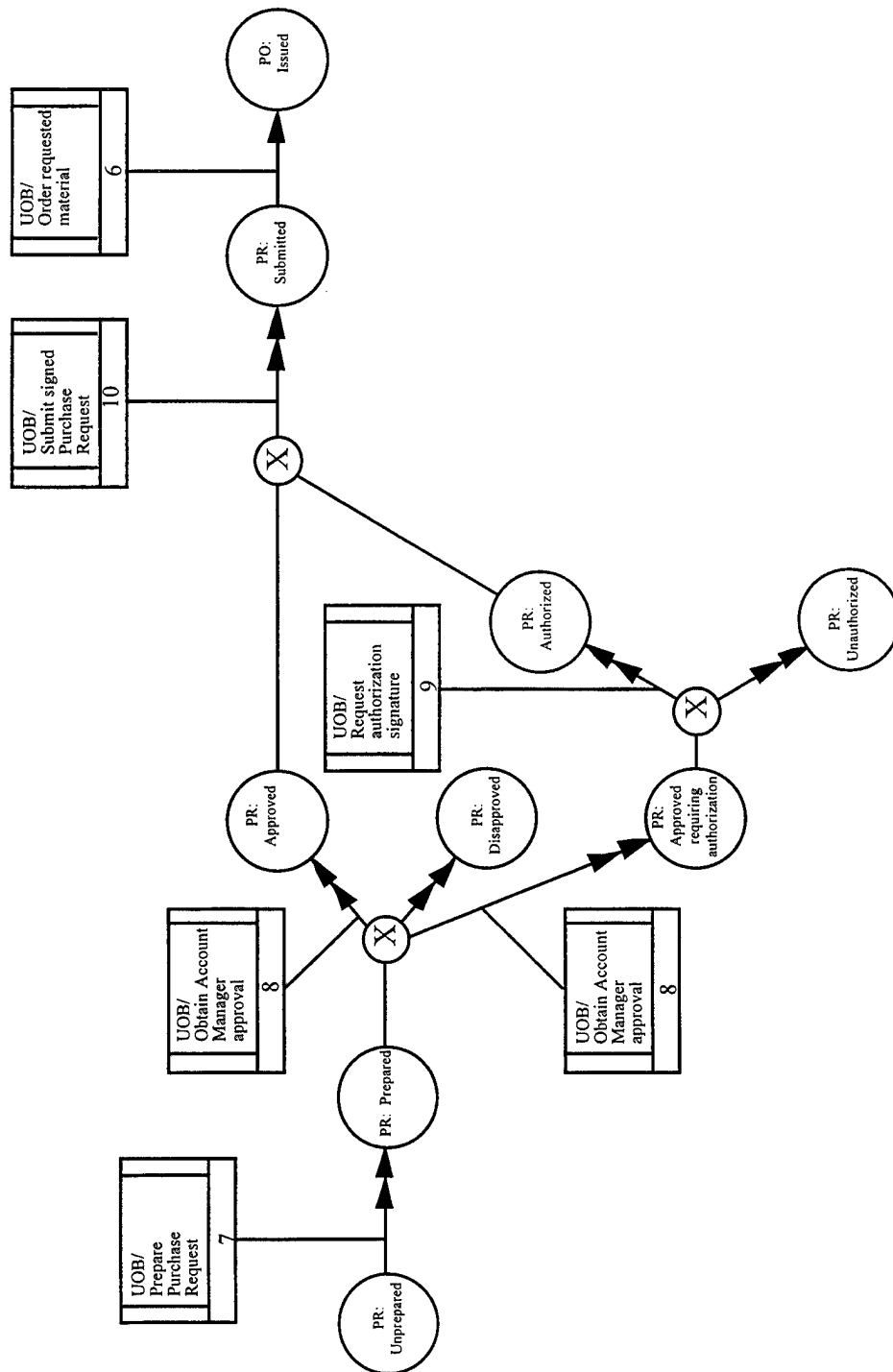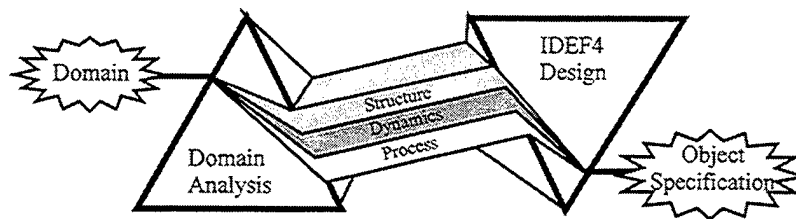
Figure A-4

Object Schematic of the Purchase Order Process

Figure A-5

IDEF4 Used Between Domain Analysis and Implementation

IDEF4 provides object oriented systems developers with a wide range of method design features, including those listed below.

1. A graphical language that makes extensive reuse of the most effective modeling conventions applied by today's object-oriented design professionals.[31]

2. Encapsulation support for maximized legacy systems reuse.

3. Mechanisms for leveraging client/server technology.

4. Support for creating reusable object-oriented designs and software components.

5. Provisions for including Commercial-Off-the-Shelf (COTS) technology in designs.

6. Design stratification support enabling reduction of design complexity.

IDEF4 supports the design-level description of the external protocols of legacy systems and COTS software. This facilitates the development of object-oriented designs that contain legacy and COTS components, resulting in object-oriented implementations that reuse existing executable software. An object-oriented design philosophy that stresses the separation of external and internal aspects of the design of an object leads to greater design success because it allows for the reuse of design components, concurrent design, design modularity, and deferred decision making.

Using IDEF4 to Develop Object-Oriented Designs. IDEF4's object-oriented design procedure works similarly to Rumbaugh's Object Method Technique (Rumbaugh, 1991) and Schlaer/Mellor's Object-Oriented

---

[31] Specific modeling conventions that were investigated and/or which contributed to IDEF4's graphical language design include Booch (1991), Rumbaugh's OMT (1991), Shlaer/Mellor (1988), Wirths-Brock (1990), Coleman, D. et al. (1994), Coad and Yourdon (1991), Martin and Odell (1992), and Jacobson's Object-Oriented Systems Engineering (OOSE) (1994).

Analysis and Design (OOA/OOD) technique ((Schlaer & Mellor, 1988) and (Schlaer & Mellor, 1992)). Table A-1 presents a comparison of IDEF4 and these methods.

There are some crucial differences Between IDEF4 and these other methods: IDEF4 is designed specifically to be compatible with other IDEF methods; IDEF4 allows the status of design artifacts to be tracked from domain object through transition to design specification and includes a design rationale component. These extra dimensions are shown in Figure A-6; the edges of the box show the progression of the design from start to finish, elaborating each of these dimensions.

In IDEF4, a design starts with the analysis of requirements and uses the domain object as input. These domain objects are encoded in their equivalent IDEF4 form and marked as domain objects. As computational objects are developed for these objects, they are marked as "transitional" and finally as "completed." The level of completion of an IDEF4 design is determined by setting measures based on the status, level, and model dimensions of individual artifacts in the design.

The system-level design starts once the "raw material" (domain) objects have been collected. This develops the design context, ensures connectivity to legacy systems, and identifies the applications that must be built to satisfy the requirements. Static, dynamic, behavioral, and rationale models are built for the objects at the system level. These specifications become the requirements on the application level — the next level of design. The application level design identifies and specifies all software components (partitions) needed in the design. Static models, dynamic models, behavioral models, and the rationale component are built for the objects at the application level. These specifications become the requirements on the next level of design — the low-level design. Static models, dynamic models, behavioral models, and the design rationale component are then built for the low-level design objects. Sublayers may be built within each layer to reduce complexity.

IDEF4 provides an iterative procedure involving partitioning, classification/specification, assembly, simulation, and repartitioning activities (Figure A-7). First the design is partitioned into objects, each of which is classified according to either existing objects or newly developed external specifications. The external specification enables the internal specification of the object to be delegated and performed concurrently. After classification/specification, the interfaces between the objects are specified in the assembly activity (i.e., static, dynamic, and behavioral models detailing different aspects of the interaction between objects are developed). While the models are being developed, it is important to simulate use scenarios or cases (Jacobsen, 1994) between objects to uncover design flaws. By analyzing these flaws, the designer can then rearrange the existing models and simulate them until a satisfactory model is produced.

Table A-1. Object-oriented Design Method Comparison (Jacobsen, 1994)

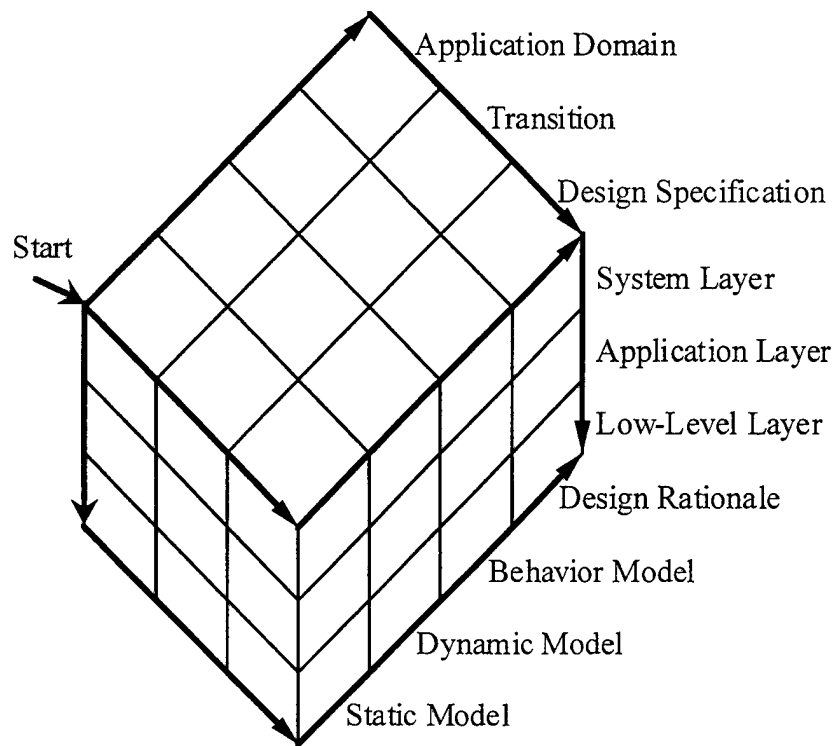|	IDEF4	Object-Oriented Analysis and Design	Object Method Technique
Encapsulation	Object	Object	Object
	Instance	Instance	Instance
	Class	Class	Class
	Partition	Subject	Subsystem
	Attribute	Attribute	Attribute
	Class Attribute	–	Class Attribute
	External Specification	–	–
Inheritance	Inheritance	Inheritance	Inheritance
	External Inheritance	–	–
	Internal Inheritance	–	–
Information Hiding	Public	–	–
	Private	–	–
	Protected	–	–
Behavior Representation	Behavior Taxonomy	–	–
	Message	Message	Message
	Method	Method	Method
	Internal Specification	–	–
Object Life Cycle	Object	Object	Object
	State	State	State
	Event	Event	Event
Relations	Link	–	Link
	Instance Link	–	–
	Client/Server	–	–
Requirements Traceability	Use Case	–	Scenario

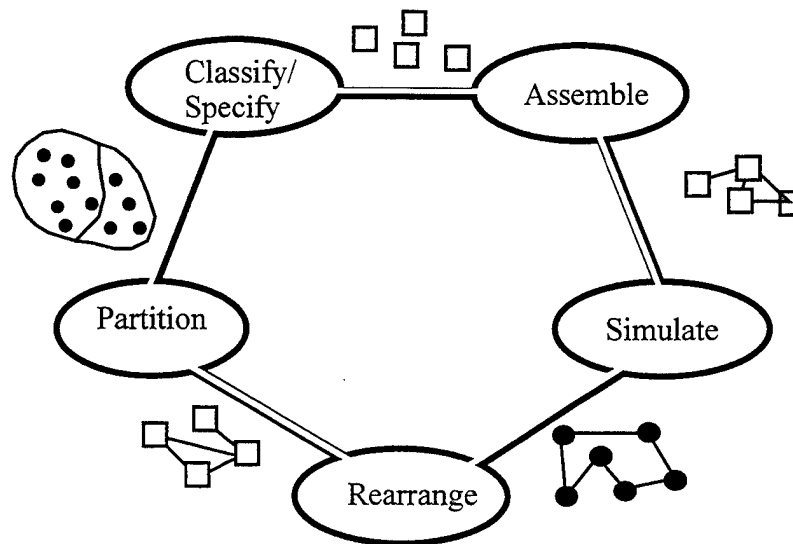**Figure A-6**

Dimensions of IDEF4 Design Objects

Figure A-7

IDEF4 Design Activities

In IDEF4, design artifacts are grouped into three models: the static model (SM), dynamic model (DM), and the behavior model (BM). The static model specifies the static structure of the design; the dynamic model specifies the dynamic communication processes between objects, classes, and systems; and the behavior model specifies the implementation of messages by methods and contains a method diagram for each message detailing the relation between the method's external behavior and implementation. Finally, the design rationale is an added component that contains major transformations of design artifacts.

Each model represents a different cross section of the design. The three design models capture the information represented in a design project; the design rationale documents the reasoning behind the design. Each model is supported by a graphical syntax that highlights the design decisions that must be made and their impact on other perspectives of the design. To facilitate use, the graphical syntax is identical among the three models. For example, in all models, the symbol "Æî indicates the artifact is derived or abstracted directly from the domain or real world, "Æ" shows the artifact is in transition, and "©" indicates the artifact has reached final design specification. In all models, boxes indicate classes, round-cornered boxes indicate instances, and arcs between boxes denote relations. No single model shows all the information encompassed in a complete design, and overlap among models ensures consistency. Each design artifact may also be associated with a formal specification of its external behavior and internal construction. The following sections discuss each model in more detail.

IDEF4 projects are organized into three design layers (Figure A-8). Each layer contains partitions, with each partition containing the three models and the design rationale component. The system layer, which represents the whole IDEF4 model, is the actual product (i.e., the application). The application layer contains the application-specific partitions or solution-specific objects. This may include actual code that has been generated and components that can be reused for other projects. The final layer is the foundation layer, which contains low-level objects, such as the buttons, forms, class libraries, and windows that constitute the software being designed.

At the system layer, the software components are industry- or domain-specific (e.g., software components related to the banking industry). As the design moves to the application layer, the software components become specialized to objects supporting task specific designs. At the foundation layer, the objects being used are software mechanisms common across a number of industries (e.g., object libraries used by the medical industry, the banking industry, and the retail industry).

Each design partition must contain the three models and the design rationale component to document the different stages and track the design at all stages of the project. The models help transition the project through these three partitions. Depending on the design situation, a project can have more or less than three design layers.

Static Model. The first model created in any of the three design partitions is the static model. This model specifies individual IDEF4 objects, object features, and the static object relationships. IDEF4 objects include instances, classes, and partitions.
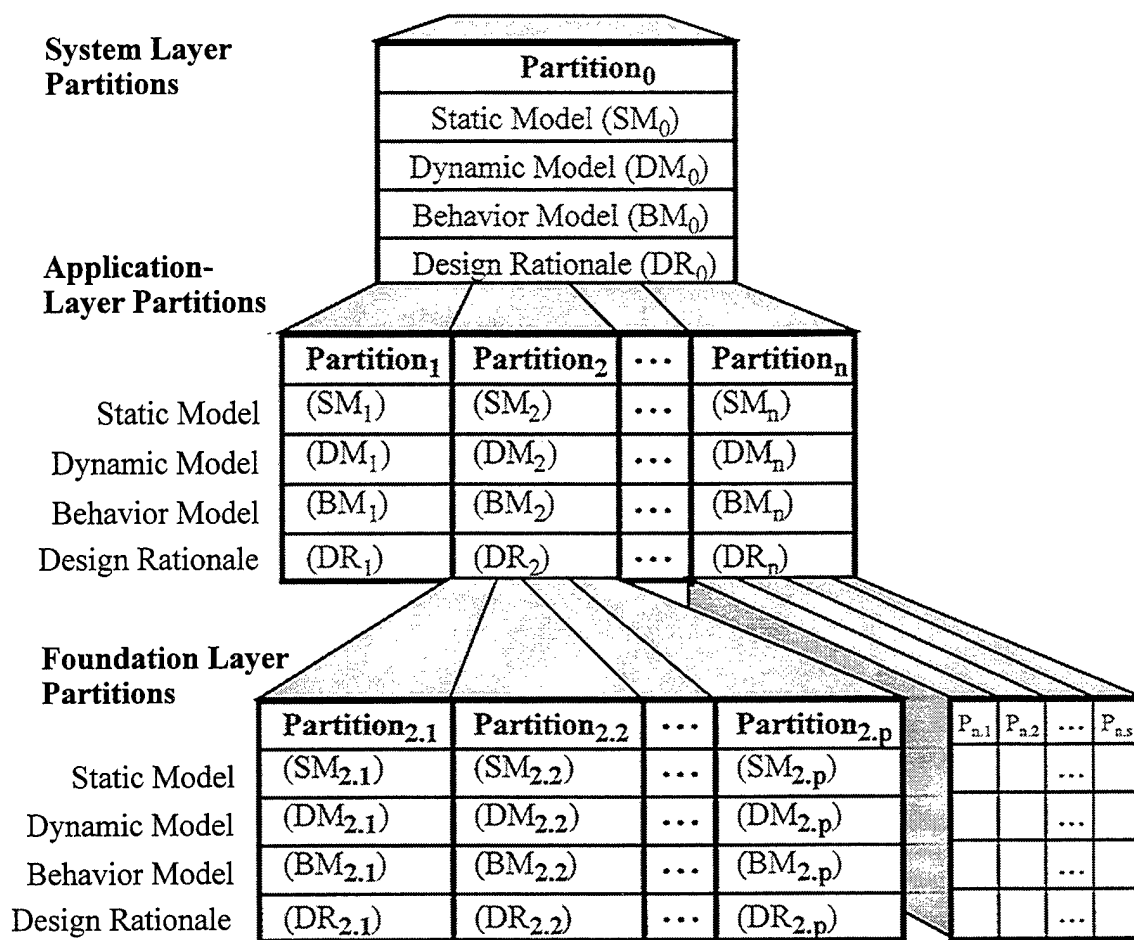
Figure A-8

Organization of the IDEF4 Project

The static model is composed of *structure diagrams*. These diagrams depict the relationships between objects and can be specialized to show specific types of relationships. Structure diagrams contain objects related by inheritance, links, and relationships, and they can be specialized to just include one type of relation structure. Specialized diagrams are labeled by the relation structure type they include (e.g., inheritance diagrams, relation diagrams, link diagrams, and instance link diagrams). However, diagrams can be created that include more than one relation type.

An *inheritance diagram* shows the class structure, specifying the inheritance relations among classes, class features, and information hiding. An inheritance diagram must contain at least one class. Figure A-9 shows a class inheritance diagram that may be used by a company to distinguish between different people entering its facility.
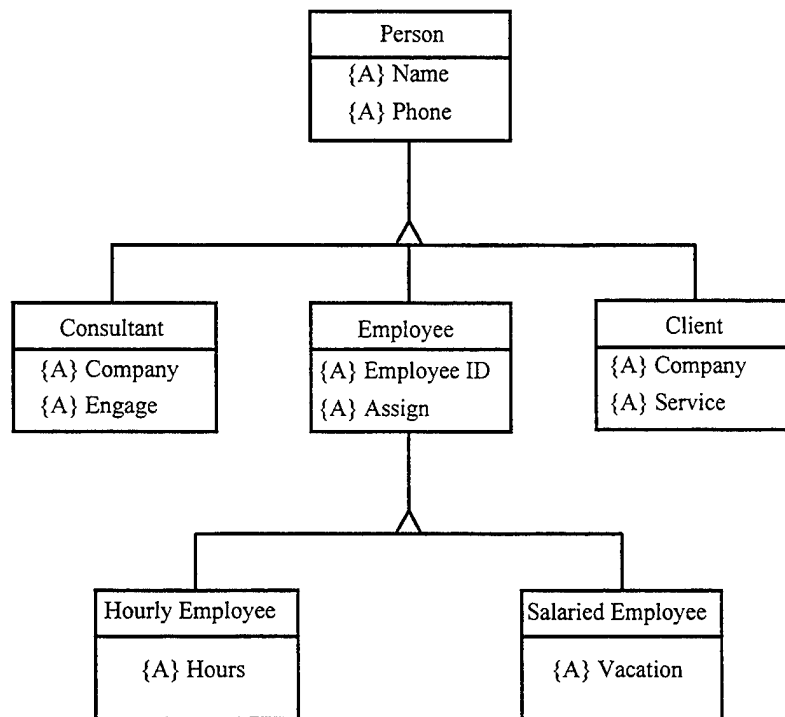
Figure A-9

Inheritance Diagram

The triangles on the relations' links point from the subclass to the superclass. In this diagram, the most general class is *Person*. The classes *Consultant, Employee,* and *Client* are specializations of *Person,* and the classes *Hourly Employee* and *Salaried Employee* are specializations of *Employee*.

Information hiding is portrayed in an inheritance diagram by sectioning the class box (Figure A-10). The features listed below the first line in the class box are public; the features listed below the public features are private features.
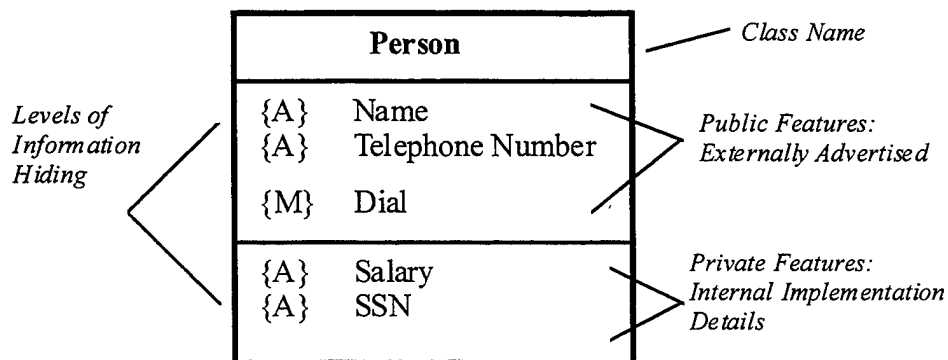

Figure A-10

IDEF4 Class Box Showing Levels of Information Hiding

Figure A-10 shows the class *Person*, which is being used in an automated telephone directory system. The person's name, telephone number, and the dial method[32] are publicly accessible, but the salary and social security number are private. If no public/private bar is shown, all features depicted are public by default. The public and private features of a class need not be shown if they do not add information to specific diagrams.

Relation diagrams focus on relation structures; that is, they display a simple relation between two objects. For example, they can show a one-to-one or a one-to-many relationship. Figure A-11 shows a relation diagram for the *Employed by/Employs* relation between *Person* and *Company*.
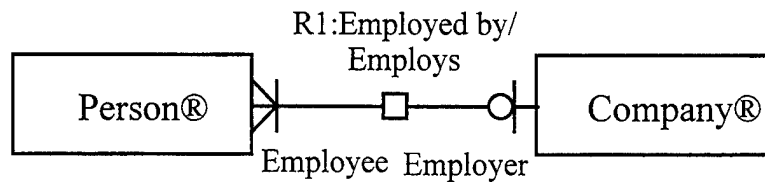

Figure A-11
Relation Diagram

In this relationship, *Person* plays the role of *employee* and *Company* plays the role of *employer*. A company may have one or more employees, and a person may be employed by zero or one companies. This cardinality constraint is shown by the circle and fork (zero or more) on the *Person* side of the relationship and by the circle and line (zero or one) notation on the Company side of the relationship.

Notice that the "®" noted after *Person* and *Company* represents an object that is from the application domain. Once this object is modified in the design evolution, the notation will change to "D."

As an object goes through design evolution, the relationships should "disappear" because the designer has implemented the relationship. The implementation will be done using links, which are depicted in a *link diagram*. Figure A-12 shows a link diagram containing a design evolution of the *Employed by/Employs* relationship to a link (L1). The link is achieved by imbedding the role names *employee* and *employer* as attributes in each class. The notation L1(R1) indicates that the link was derived from relation R1. Links in IDEF4 may be implemented by pointers, indices, or foreign keys. The imbedded attribute is known as a referential attribute because it refers to other object instances. Notably, the link notation is brought into the class box with a circle on the end to indicate that the relation has been embedded.

---

[32] The term "method," as used here, refers to a specification of class behavior and is consistent with use of the term in the object oriented systems development domain. This use of the term differs from that applied when discussing IDEF4 or other IDEF methods.
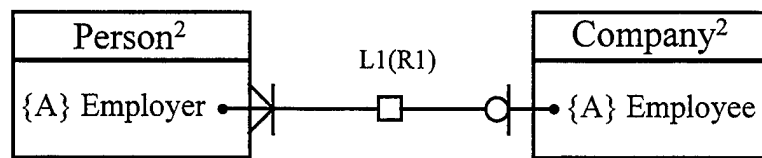
Figure A-12

Link Diagram

The evolution from relation to link is tracked by the L1(R1) link label. There may be other constraints on the relationship R1 that will constrain methods that create, reference, update, and delete entities in the domain of the attributes defining the link. The IDEF4 method helps the project team keep track of design evolutions like this.

The validity of the link diagram is tested using the *instance link diagram*. Object instances are run through the design describing real or possible object relationships. Figure A-13 shows an instance link diagram with two instances of the class *Person*, [*John*] and [*Mary*], using the *Employed by/Employed* relation link *L1*. [John] and [Mary] work for the [XYZ] company.

Figure A-13

Instance Link Diagram
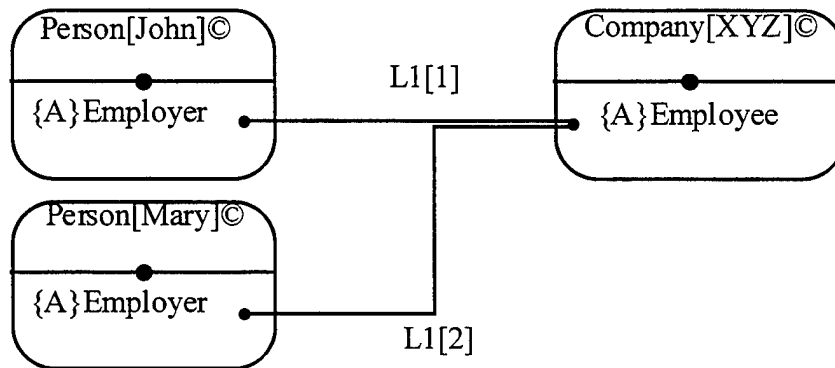
Object instances are represented by round-cornered boxes with a black dot in the center. Link instances are represented by connections with a black dot. The black dot serves as a visual cue for instances and is used in the IDEF3 Process Method and the IDEF5 Ontology Method. The instance link diagram is useful for challenging constraints in the design by exploring boundary conditions.

215

Behavior Model. The behavior model consists of *method diagrams*. Method diagrams show polymorphism[33] and are used to take advantage of behavioral similarity by reusing method specifications. Figure A-14 shows a method diagram highlighting the *Pay* method for employees. Permanent employees are provided benefits with every paycheck while temporary employees receive pay with no benefits. Consequently, the method for calculating payment for permanent employees is different from the method for calculating payment for temporary employees. The diagram highlights the difference between messages and the methods that implement the messages. For example, the message *Pay* is implemented by the method *BenefitPay* for permanent employees and by *StraightPay* for temporary employees.
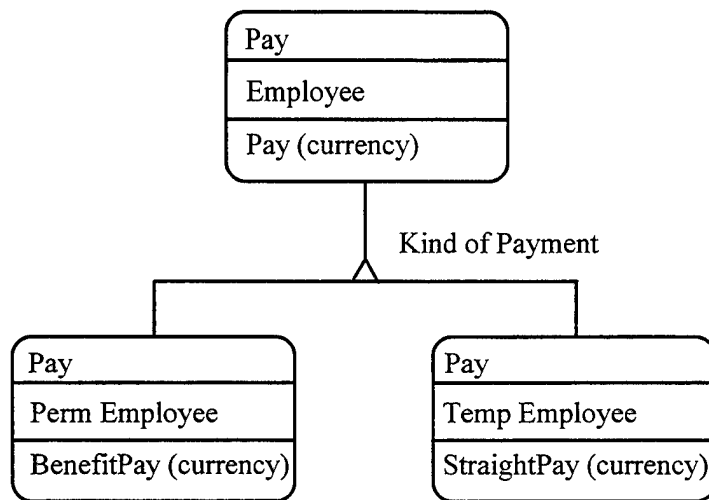
Figure A-14

Behavior Diagram

The behavior diagram, which provides a view of a class of behavior across all classes, is useful for maintaining a consistent method signature (external protocols) or behavior across classes. The triangle on the relationship line denotes a generalization/specialization relationship which indicates that the behavior of *Perm_Employee.Pay* and *Temp_Employee.Pay* specializes the behavior and implementation of *Employee.Pay*.

Dynamic Model

The dynamic model has two kinds of diagrams, *client/server diagrams* and *state diagrams*. Both diagrams use asynchronous and synchronous communications to depict dynamic relations between objects. The dynamic relations are modeled using events and message passing.

---

[33] The term polymorphism, when applied to object-oriented systems, means that different objects may respond differently to the same message.

Client/server diagrams illustrate the way objects use one another. IDEF4 uses client/server diagrams as a common syntax for describing the usage relationships between objects — from the system level of abstraction to low-level objects. Figure A-15 shows a client/server diagram depicting the dynamic payment relationship between *Calendar*, *Payroll*, *PermEmployee*, and *TempEmployee*.
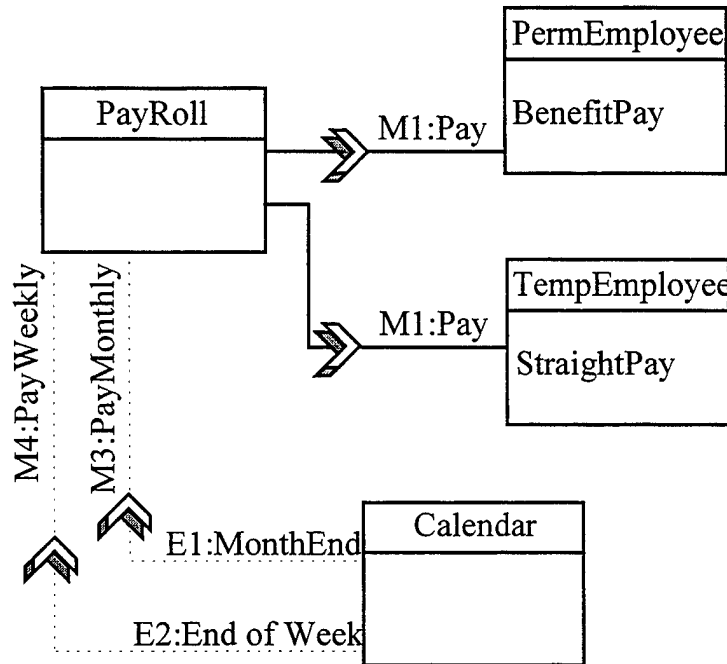

Figure A-15

Client/Server Diagram

The calendar generates events for month ends and week ends. Payment by *Payroll* is initiated by MonthEnd and End of Week events generated by *Calendar*. (The dashed arc indicates asynchronous communication between *Calendar* and *Payroll*.) The chevron on each link points in the direction the event or message is being sent; shadowed chevrons indicate parameters are included. *Payroll* sends pay messages to permanent employees and temporary employees. The *Pay* message is synchronous — as depicted by the solid line — and is implemented by the BenefitPay method in *PermEmployee* and by the StraightPay method in *TempEmployee*.

State diagrams document the relevant states of the object and the permissible state transitions. A state represents a situation or condition of the object during which certain constraints apply. (The constraints may be physical laws, rules, policies, and so forth.) The transitions are modeled using events and actions. Events are triggers which cause the object to initiate a transition from one state to another. The action is initiated upon entry into a state (Figure A-16).
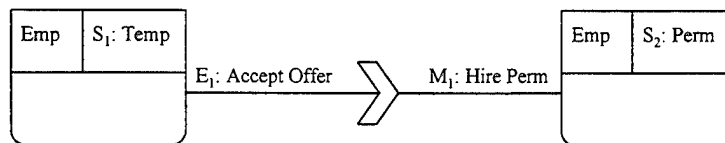
Figure A-16

Employee State Diagram

The state diagram depicts the behavior of an object by showing the states in which an object can exist and how the object transitions from state to state. Using the example from the client/server diagram, an employee who is temporary may be hired on as a permanent employee. The TempEmployee object would have the method *Hire Permanent*. The "accept" event generated by a temporary employee is depicted on the relationship line. The message associated with this event, "Hire Permanent," is depicted on the other end of the relationship line.

Design Rationale Component. An aspect of IDEF4 that sets it apart from other object-oriented design methods is its ability to distinguish explicitly between real-world objects, evolving design objects, and final design objects and to record the evolution between these objects. The design rationale component of IDEF4 records major transitions in the evolution of an IDEF4 design. IDEF4 documents major design milestones as design states by recording the participating diagrams as a design state, then giving the rationale for transition to another design state. The rationale is captured by describing the triggering observations and the resulting actions. Figure A-17 shows three design states. Observations are normally symptoms or concerns about the design.
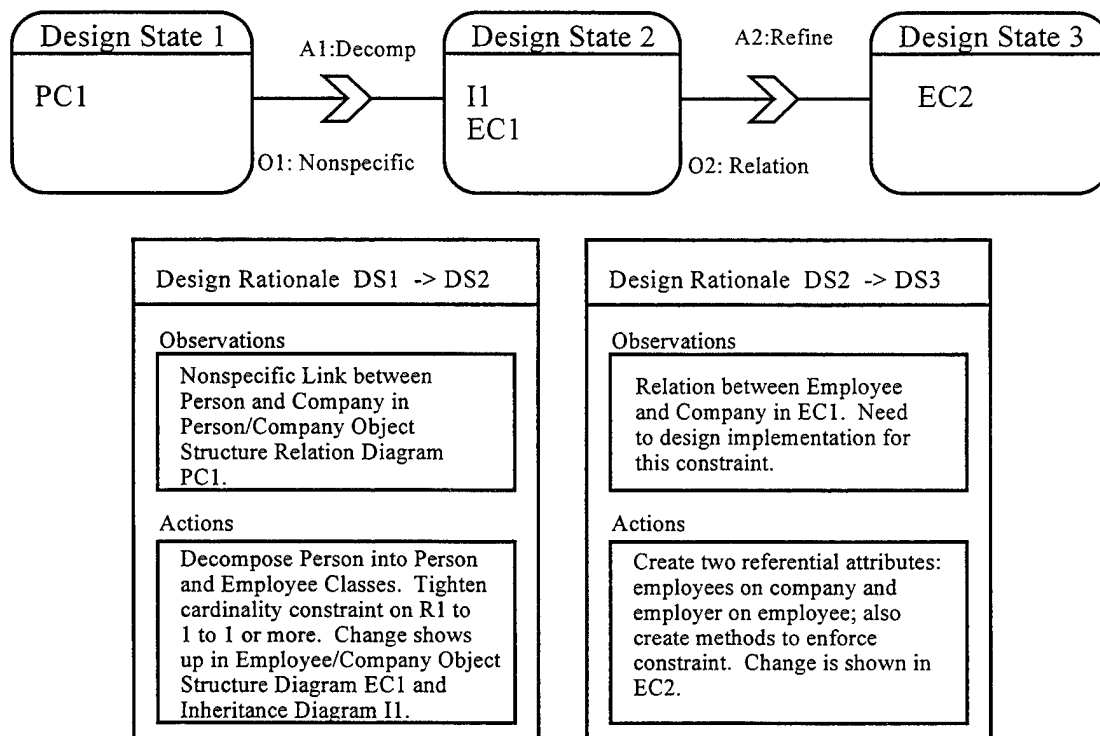

Figure A-17

Design Rationale Diagram

218

Each design state lists the associated diagrams. The transition arc between design states has a set of observation events from a design state and a set of actions that is applied to the next design state. Thus, rationale diagrams are actually specialized state diagrams.

IDEF4 allows users to document detailed specifications associated with any design artifact. There are two kinds of design specifications: external design specifications (ExSpec) and internal design specifications (InSpec). ExSpec defines the behavior of design features (Signature), whereas InSpec defines how the design object is to achieve that behavior.

A designer reusing a design object follows the external specification; a software engineer implementing the design follows the internal specification. IDEF4 promotes the view of software engineers as software component builders or software component users. The ExSpec enables the creation of component design libraries for design reuse.

Summary. IDEF4 is a method that has been developed to:

- apply object-oriented design techniques in the context of the IDEF family of methods;

- separate an object's external and internal design specifications, design systems which can interface to legacy systems and commercial systems;

- employ and update the design during system use and maintenance;

- reuse design objects in other designs; and

- specify object-oriented, distributed computing environments.

The reuse of design artifacts, software objects, and executable components is facilitated by syntactic support for encapsulating components. The maintainability of design and software components is enhanced by emphasis on documentation and artifact traceability.

A more detailed description of the IDEF4 method can be found in the *IDEF4 Object-Oriented Design Method Report* [(Mayer et al., 1992) and (Mayer et al, forthcoming)].

IDEF4/C++ Object-Oriented Design Method Overview

IDEF4/C++ is an extension of the generic IDEF4 Object-Oriented Design method, developed and specialized for users of the IDEF4 method who intend to create C++ implementations for their IDEF4 conceptual designs. IDEF4/C++ specifies design objects with sufficient detail to enable C++ source code generation. IDEF4 and IDEF4/C++ provide a smooth evolution between domain analysis, conceptual design, and system implementation in C++ (Figure A-18).
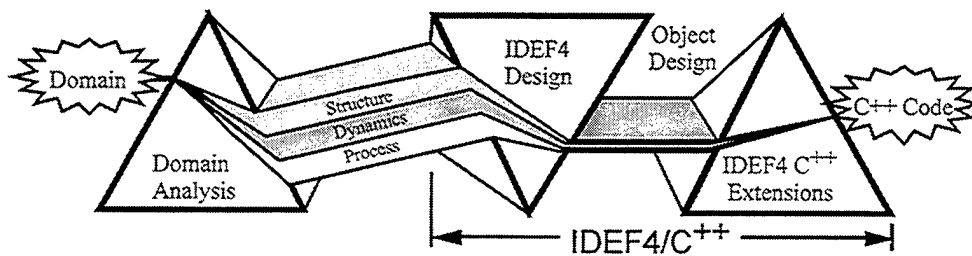
Figure A-18

IDEF4 and IDEF4/C++ Used Between Domain Analysis and C++ Code

IDEF4/C++ was designed to perform the following tasks:

1. Maintain consistency in notation and practice with the IDEF4 method.

2. Provide graphical notation to support C++-specific constructs.

3. Provide for consistent transition from conceptual design to C++ code.

4. Support legacy system reuse through object-oriented encapsulation.

5. Help leverage client/server technology.

6. Promote the development of reusable C++ software components.

7. Facilitate use of COTS technology in designs.

8. Reduce design complexity through design stratification.

Like IDEF4, IDEF4/C++ supports the design-level description of the external protocols of legacy systems and COTS software. This facilitates the development of object-oriented designs that contain legacy and COTS components and results in object-oriented implementations that reuse existing executable software. An object-oriented design philosophy that stresses the separation of external and internal aspects of the design of an object leads to greater design success because it allows for the reuse of design components, concurrent design, design modularity, and deferred decision making.

The IDEF4/C++ multidimensional approach to object-oriented software system design consists of the following items:

1. Design layers (system-level, application-level, and low-level design).

220

2.   Artifact design status[34] (application domain, in transition, design specification, and C++ code specification).

3.   Design models (static, dynamic, and behavior) and the design rationale component.

4.   Design features, ranging from general to specific, enabling deferred decision making.

In other words, IDEF4/C++ divides the object-oriented design activity into discrete, manageable units. Each subactivity is supported by a graphical syntax that highlights the design decisions that must be made; their impact on other perspectives of the design are revealed by navigating through the diagrams. No single diagram shows all the information contained in the IDEF4/C++ design model, thus limiting confusion and allowing rapid inspection of the desired information. Carefully designed overlap among different types of diagrams ensures compatibility between the different submodels.

Using IDEF4/C++ to Develop C++-Targeted Object-Oriented Systems. IDEF4/C++ is an object-oriented design method providing a multimode approach to transforming application domain information to design specifications (Figure A-19). IDEF4/C++ focuses on the design process, relying on IDEF domain analysis methods or other methods to supply models of the application domain. The IDEF4/C++ process outputs design specifications to an implementation process. The major activities in IDEF4/C++ object-oriented design are requirements analysis, system design, application design, and low-level design.
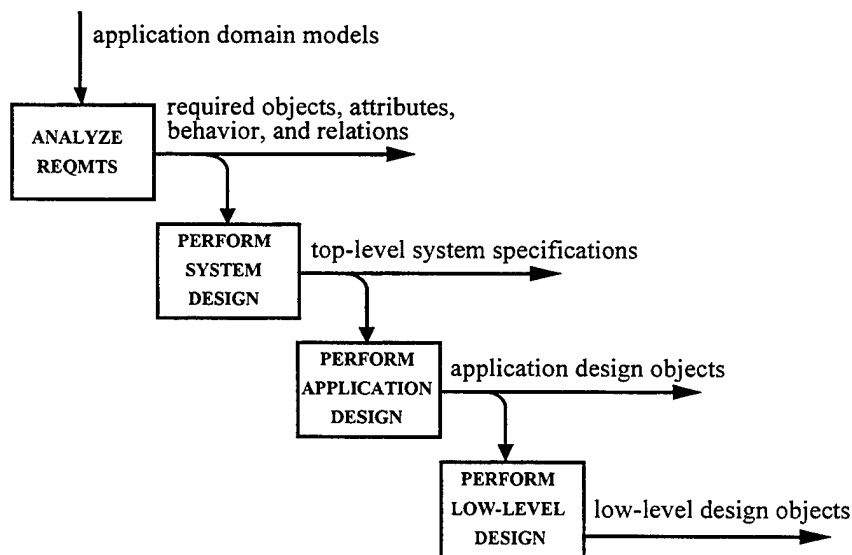

Figure A-19

The Modes of IDEF Object-Oriented Design

---

[34] IDEF4/C++ adds C++ code specification to IDEF4's artifact design status categories.

The requirements analysis activity processes information from domain models and requirements documents to establish clear, concise requirements for the other activities. The requirements analysis activity provides both initial objects and use scenarios. The use scenarios map to requirements and are used to validate that requirements are satisfied in design and implementation.[35] The system design activity establishes design strategies, divides the design into top-level design partitions, and defines the interfaces with other systems. The application design activity specifies the object-oriented design details in these partitions. The low-level design activity generates language-specific implementation specifications.

The design process is the predominant activity of the early stages of the software system development life cycle. The design artifacts resulting from an IDEF4/C++ design activity are important to both C++ implementation and the subsequent sustaining activities. The following material discusses the planning and implementation involved in system development, and the IDEF4/C++ method's role in the design/implementation life cycle.

According to Ramey's System Development Methodology (SDM) (Ramey, 1983), there are two developer roles in system development: technical and administrative (Figure A-20). Developers in the technical role are concerned with technical excellence. They do not concern themselves with resource constraints, such as time, money, and the like. On the other hand, developers in the administrative role are concerned about budget and scheduling constraints. Understandably, the conflicting concerns of each role view is the source of much tension.

Technical Role	Administrative Role
Technical Excellence	Business Case
Technical Merit	Return on Investment (ROI)
The Best Product	Organizational Goals
Creativity	Standards
Random Life Pattern	Life Cycle

Figure A-20

Technical and Administrative Roles

---

[35] Use scenarios correspond to use cases and use-case analysis in Jacobsen's OOSE (Jacobsen, 1994).

The theory of design life cycle helps create an understanding of the basic design processes, particularly for administrative purposes. The design process from such a view is assumed to be cyclical, with steps beginning at some point, continuing through maturity, and eventually ending.

This view of design as a series of incremental and sequentially interdependent steps is an attempt to order the steps of the process so each step becomes independent, except for its obvious relation to the steps immediately preceding and following it. Design strategies can be considered "meta-plans" for addressing the complexities of frequently occurring design situations (e.g., methodizations or organizations of design activities). Three types of design strategies can be considered:

1. **External-Constraint-Driven Design** — Design for situations in which the software goals, intentions, and requirements are not well-characterized, much less defined. These situations often result when the designer is brought into the product development process too early.

2. **Characteristic-Driven Design** — Software design in a closely controlled situation for which strict accountability and proof of adequacy are rigidly enforced. These design situations often involve potentially life-threatening situations.

3. **Carry-Over-Driven Design (Routine Design)** — Changes to existing, implemented designs or designs that are well understood (e.g., sorting).

From an OOD point of view, the external-constraint-driven and carry-over-driven strategies are the most common design situations. The design development procedure, outlined in the following sections, is a distillation of the experience and insights gained in building several IDEF4/C++ designs on case studies for situations requiring these design strategies.

The first step in a design strategy is determining the developer focus. The type and stage of the design dictates how much emphasis is put on the designer as an administrator or technical developer. The roles should be decided and implemented in the strategy to manage expectations and prevent differing goals for the design.

From the administrative perspective, development is a business venture. It is a very structured undertaking that occurs over a fixed period of time and within fixed resource limitations. The system development process is characterized by line items and milestones. The administrative developer manages contractual requirements (e.g., schedules, budgets, delivery items), company goals, and guidelines. Trade-offs are made between the constraints of costs, schedules, and technical excellence.

The technical developer, on the other hand, works to discover the problem, develop a solution, create a product, and implement a system. Constraints are viewed in terms of development tools, not time and cost. A technical developer may review the discovery of the problem and solution multiple times without ever progressing to the creation of an actual product (Figure A-21).
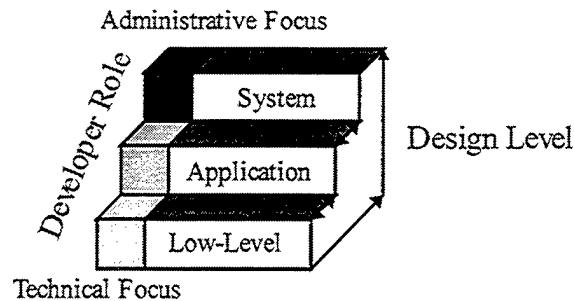

Figure A-21

Developer Focus at the Design Level

A design strategy should include administrative and technical viewpoints. If the software is consumer-oriented, administrative development characteristics are more applicable and should be weighted more heavily than the technical development characteristics. In a research and development environment, the conditions are inverted. However, in both situations, the technical development view also should be weighted more heavily in the initial stages of design. Furthermore, as the design solidifies and the product becomes tangible, the focus should switch to the administrative characteristics.

Developing IDEF4/C++ designs is a process of specifying the structure and behavior of an object-oriented program or system in C++. The process uses design layers containing partitions which help manage the complexity of the design. The design layers address the administrative role of a developer. A design layer has five design activities that evolve the design from the initial stages to C++ implementation. These five design activities address the technical role of the developer.

The IDEF4/C++ method uses a multidimensional approach to object-oriented software system design (Figure A-22). These dimensions are design layers (requirements analysis, system, application, and low-level design), artifact design status (application domain, in transition, design specification, and C++ specification), and design models (static models, dynamic models, behavior models, and rationale component).[36] The design models

---

[36] Setting criteria for completion of the design is important in the design process. This is particularly true of modeling situations to prevent never-ending model development. It is not possible to give precise criteria for the completion of design activities, but as the design approaches completion, the rate of change of the design will

may be developed or reused from other projects. The method has explicit syntactic support for these dimensions, and the design procedure is organized around the layers or levels of design.
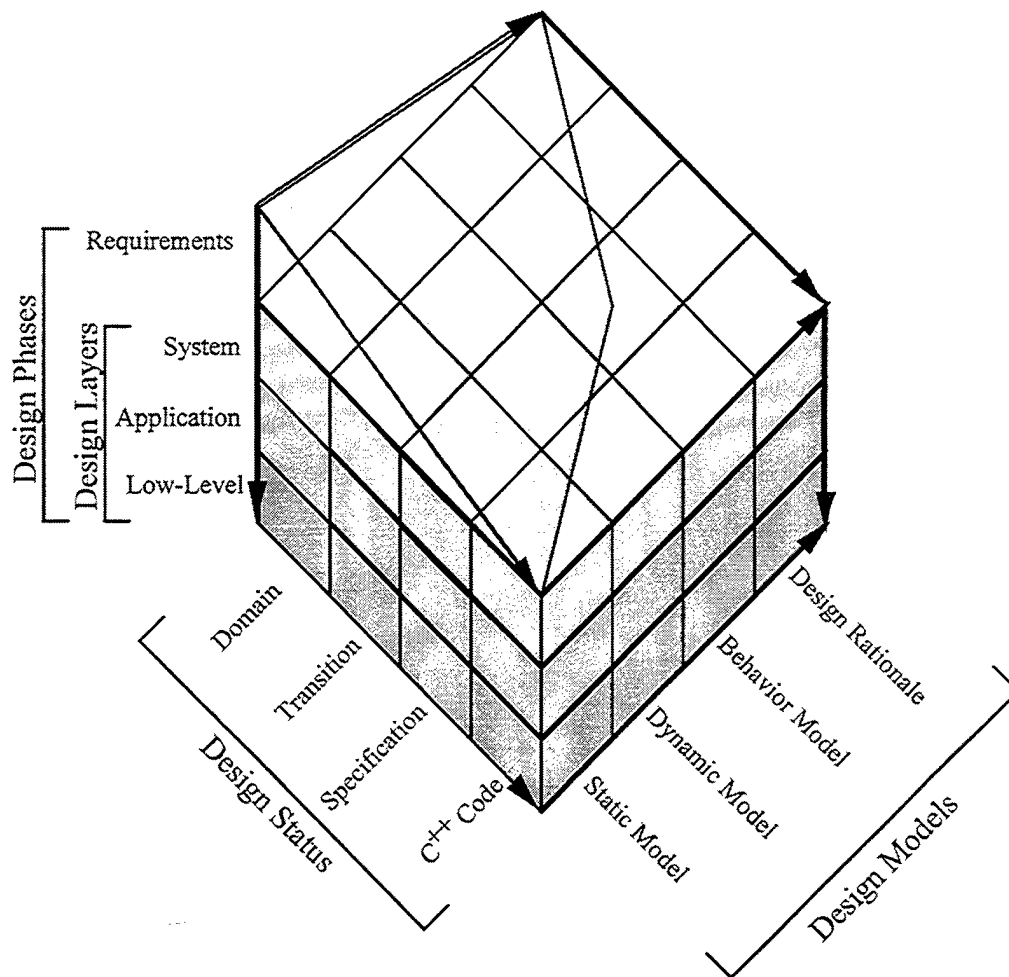
Figure A-22

IDEF4/C++ Design Layers Relative to Model and Status

Each layer (system, application, and low-level) contains a static model, dynamic model, behavior model, and design rationale. These layers can be viewed as steps in the design process because they track the design from the highest level of abstraction (system-level) to the actual software components (low-level). The role of the developer will vary during the different phases of design based on the type of design being created. Focusing on an administrative or technical view will also affect the products resulting from the design activities.

decrease. This occurs because each new design constraint that is specified places more and more restrictive requirements on the design. When this occurs, model development should taper off.

When using IDEF4/C++ to develop an object-oriented design, the following five general design activities[37] are applied recursively[38] to each layer:

1. Partition — Partition the evolving design into smaller, more manageable pieces by like behavior. In this activity, objects are identified.

2. Classify/Specify — Classify the design against existing definitions or specify a new design object. In this activity, object behavior (methods, state) is defined and objects are compared and contrasted with each another. This activity is responsible for C++ code specification in IDEF4/C++.

3. Assemble — Incorporate design objects into existing structures. In this activity, relations, links, and dynamic interactions among objects are identified. This activity is also responsible for compiling and linking[39] when the design reaches C++ specification.

4. Simulate — Validate[40] and verify[41] the models by exercising them against use scenarios in the requirements model. When the design reaches C++ specification, the code may be validated and verified by testing the executable code against use scenarios. During this activity, problems with the current design/implementation are identified and described.

5. Rearrange — Rearrange existing structures to take advantage of newly introduced design objects. This may result in an adjustment in the design partitioning which would cause another iteration in the design.

---

[37] The sequence of the design activities is not implied by the order in which they appear.

[38] The notion of recursive application is used to describe how the process of specification is applied to each element in partitioning the five design types as well as to the overall design development activity. This recursion continues until the prototype classifications of the resulting elements can be clearly established.

[39] Compiling and linking produces executable code from the C++ source code.

[40] Validation involves determining if the system satisfies requirements.

[41] Verification involves determining if the system behaves as intended.

The five design activities described above will be applied iteratively to each design layer. The design layers are viewed as design phases, which move the project from the initial stages to the final product while simplifying the management of the design.

These five design activities are organized into four phases of effort:

Phase 0 —Requirements Analysis

Phase 1 — System Design Layer Development

Phase 2 — Application Layer Development

Phase 3 — Low-Level Design Layer Development

An IDEF4/C++ design starts with requirements analysis (Phase 0) and results in a requirements model. The requirements model consists of a function model (IDEF∅) used to scope the effort and use scenarios (IDEF3).Elements in these models are mapped to requirements so that they can be used to validate the models. Domain objects identified during the requirements analysis phase are input to the IDEF4/C++ design. They are encoded in IDEF4/C++ form and marked as domain. As computational objects are developed, they are marked as transitional. As the design solidifies, they become C++ objects. The level of completion of an IDEF4/C++ design is determined by the status of individual artifacts in the design (i.e., as the objects in the design stabilize, the design itself stabilizes).

The other three distinct design phases are: system layer design, application layer design, and low-level design. Structuring the design process into these design layers reduces the complexity of the design. The system layer design activities (Phase 1) ensure connectivity to other systems in the design context. The application layer activities (Phase 2) focus on the interfaces between the components of the system. These components include commercial applications, previously designed and implemented applications, and applications to be designed. The low-level design layer tasks (Phase 3) are responsible for the definition of the foundation objects of the system.

In each of these phases, IDEF4/C++ prescribes an iterative procedure in which the design is partitioned into objects. External specifications are developed for each object so that the object's internal specification may be done concurrently. After partitioning, static, dynamic, and behavioral models detailing different aspects of the interaction between objects are developed. After the models are developed, it is possible to simulate interaction scenarios between objects to uncover design flaws. The existing models are then rearranged and simulated/executed until a satisfactory model is produced.

The final models which result from the IDEF4/C++ design evolution process normally consist of three layers, each containing the three main submodels: static model, behavior model, and dynamic model. From the

(final version of the) foundation (low-level) layer, the transition from these submodels to implementation code is relatively straightforward. Inheritance and link diagrams in the static model provide the C++ programmer with the class hierarchy. The IDEF4 class boxes in the inheritance diagrams map directly to C++ class definitions, and the referential attributes (within a class) from the link diagrams map directly to pointers to classes or objects in C++. Behavior diagrams from the behavior model provide information about method reuse between classes, both for internal and external specifications.

Finally, the client/server and state diagrams coupled with the action and event specification forms from the dynamic model provide implementation details for methods. An example of the transition from state diagrams to method implementations is presented in Figure A-23.
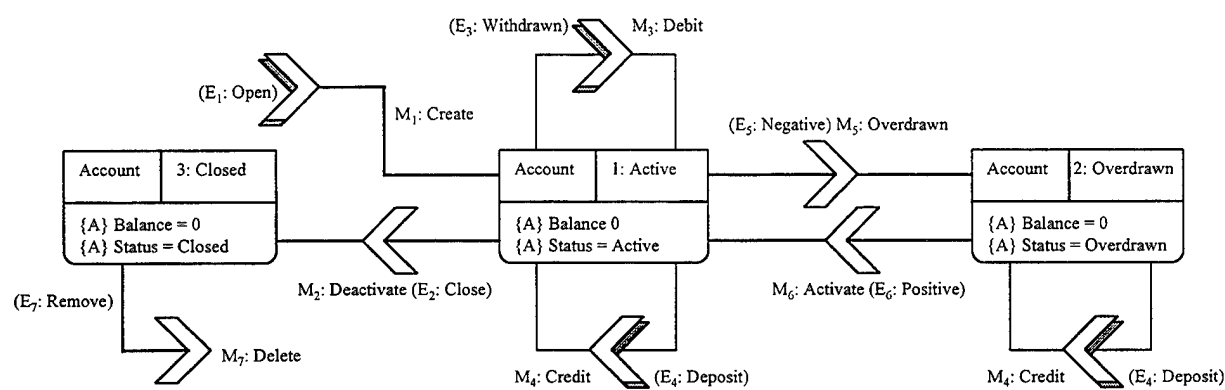
Figure A-23

State Diagram for a Bank Account

From the state diagram and the designations for the states, the implementations of some methods can be constructed. For example, for the message *Deactivate* which is sent from *Account* to *Account* to transition from state *Active* to state *Closed*, the C++ implementation for the *Deactivate* method would be as follows:

```
void Account::Deactivate(void)

{

    Balance = 0;      // set Balance to zero

    Status = Closed;  // close the account

}
```

Other method implementations would be constructed from the information in the state diagrams in tandem with the information from the event and action specification forms. For example, the internal event specification for the event E5 is:

ID	Name	Description	Condition
E5	negative	Balance of active account becoming negative.	S_1 and Balance < 0

This information provides the context in which the *Overdraw* method will be called (i.e., the *Overdraw* message will be sent to *Account* in response to the balance dropping below zero and the account being in state S1, where S1 is the *Active* state). The action specification for the message M5, which will be implemented as the method *Overdraw* in the class *Account*, is shown below.

ID	Name	Description	Specification
M5	overdraw	Flag account as overdrawn.	overdraw() {Balance = Balance - Penalty;}

These two specifications provide sufficient detail to implement the method *Overdraw* as shown in the following C++ code:

```
void Account::Overdraw( void )

{

    Balance -= Penalty;    // subtract the penalty (fee) for overdrawing

    Status = Overdrawn;  // set the status to Overdraw

}
```

which is called from within the method Debit as shown:

```
void Account::Debit(float Amount)

{

    Balance -= Amount;  // debit the account

    if( Balance < 0 )      // check for negative balance

        Overdraw();         // if negative, call Overdraw method

}
```

The remainder of the methods in *Account* would be implemented similarly. The general technique for transitioning from the dynamic model to C++-method implementations consists of examining the state diagrams and the event and action specification forms. Additional insight, when needed, can be obtained by examining the client/server diagrams to obtain a slightly higher-level perspective than that provided by the state diagrams.

Summary. IDEF4/C++ has been developed to design C++-based systems; apply object-oriented design techniques leveraging the IDEF family of methods; separate an object's external and internal design specifications; design systems which can interface to legacy systems and commercial systems; employ and update the design during system use and maintenance; reuse design objects in other designs; and specify object-oriented, distributed computing environments. The final models which result from the IDEF4/C++ design evolution process normally consist of three layers of models representing system layer design, application layer design, and low-level design; respectively. Each layer will contain the static model, the behavior model, the dynamic model, and a design rationale component. The transition from these submodels to implementation code is relatively straightforward. Reuse of design artifacts, software objects, and executable components is facilitated by syntactic support for encapsulating components. The maintainability of design and software components is also enhanced by emphasis on documentation and artifact traceability with the resulting C++ code.

A more detailed description of the IDEF4/C++ method can be found in the *IDEF4/C++ Object-Oriented Design Method Report* (Mayer et al., 1995b).

IDEF5 Ontology Description Capture Method Overview

Historically, ontologies arose from the branch of philosophy known as *metaphysics*, which deals with the nature of reality — of what exists. The traditional goal of ontological inquiry, in particular, is to divide the world "at its joints," to discover those fundamental categories or *kinds* that define the objects of the world. So viewed, natural science provides an excellent example of ontological inquiry. For example, a goal of subatomic physics is to develop a taxonomy of the most basic kinds of objects that exist within the physical world (e.g., protons, electrons, muons). Similarly, the biological sciences seek to categorize and describe the various kinds of living organisms that populate the planet. Abstract sciences and mathematics, in particular, attempt to discover and categorize the domain of abstract objects such as prime numbers, polynomial algorithms, commutative groups, topological spaces, and so forth. Additional examples of ontological inquiry can be observed in the fields of geology, psychology, chemistry, sociolinguistics, and, in general, any discipline that attempts to understand the nature of some set of physical, psychological, or social phenomena.

The natural and abstract worlds of pure science do not exhaust the applicable domains of ontology. There are vast, human-designed and human-engineered systems such as manufacturing plants, businesses, military bases, and universities in which ontological inquiry is just as relevant and just as important. In these human-created systems, ontological inquiry is primarily motivated by the need to understand, design, engineer, and manage such

systems effectively. Consequently, it is useful to adapt the traditional techniques of ontological inquiry in the natural sciences to these domains as well.

Ontological analysis is accomplished by examining the vocabulary that is used to discuss the characteristic objects and processes that compose the domain, developing rigorous definitions of the basic terms in that vocabulary, and characterizing the logical connections among those terms. The product of this analysis, an *ontology,* is a domain vocabulary complete with a set of precise definitions, or *axioms*, that constrain the meanings of the terms sufficiently to enable consistent interpretation of the data that use that vocabulary.

An *ontology* includes a catalog of terms used in a domain, the rules governing how those terms can be combined to make valid statements about situations in that domain, and the *sanctioned inferences* that can be made when such statements are used in that domain. In every domain, there are phenomena that the humans in that domain discriminate as (conceptual or physical) objects, associations, and situations. Through various language mechanisms, we associate definite descriptors (e.g., names, noun phrases, etc.) to those phenomena. In the context of ontology, a *relation* is a definite descriptor referring to an association in the real world; a *term* is a definite descriptor that refers to an object or situation-like thing in the real world.

In constructing an ontology, we try to catalog the descriptors (like a data dictionary) and create a model of the domain, if described with those descriptors. Thus, in building an ontology, you must perform three tasks: (1) catalog the terms; (2) capture the constraints that govern how those terms can be used to make descriptive statements about the domain; and (3) build a model that, when provided with a specific descriptive statement, can generate the "appropriate" additional descriptive statements. The expression *appropriate descriptive statements* means two things. First, because there are generally a large number of possible statements that could be generated, the model generates only the subset that is "useful" in the context. Second, the descriptive statements that are generated represent facts or beliefs typically held by an intelligent agent in the domain who had received the same information. The model is then said to embody the *sanctioned inferences* in the domain. It is also said to *characterize* the behavior of objects and associations in the domain. Thus, an ontology is similar to a data-dictionary but includes both a grammar and a model of the behavior of the domain.

The benefits of ontology development can be grouped under two headings: benefits of developing the ontology and benefits derived from the products of ontology development.

1. **Benefits of Developing the Ontology:** Ontological analysis is a discovery process that leads to an enhanced understanding of a domain. The insights gained through ontological analysis are useful for identifying problems (diagnosis), determining the cause of problems (causal analysis), formulating alternative solutions (discovery and design), building consensus, and knowledge sharing and reuse.

2. **Benefits Derived from the Products of Ontology Development:** The ontologies produced through ontological analysis can be used beneficially for information systems development. That is, ontologies provide a blueprint for developing more intelligent and integrated information systems. In terms of systems development, ontologies can be used as reference models for planning, coordinating, and controlling complex product/process development activities. Furthermore, as a tool of business process re-engineering, ontologies provide clues to help identify focus areas for organizational restructuring and suggest potential high-impact transition paths for restructuring.

IDEF5 method development was undertaken to fill a gap in the existing set of methods. A type of information — ontological information — has not been targeted directly by any existing method. Additionally, ontology development has traditionally been a difficult and expensive task. Ontologies developed to date, such as TACITUS (Hobbs, Croft, Davies, Edwards, & Laws, 1987) and CYC (Lenat, Prakash, & Shepherd, 1986), are the result of very expensive handcrafted efforts by highly skilled specialists. Many enterprises are unable to fund such expensive efforts. However, efforts such as these demonstrated that a standard and cost-effective means of developing ontologies was needed for enterprises to leverage the benefits of ontology development.

The IDEF5 method provides a theoretically and empirically well-grounded *method* specifically designed to assist in creating, modifying, and maintaining ontologies. Standardized procedures, the ability to represent ontology information in an intuitive and natural form, and higher quality results enabled through IDEF5 application also serve to reduce the cost of these activities.

Using IDEF5 to Develop Ontologies. The IDEF5 ontology development process consists of the following five activities. Specific guidelines and techniques are provided to help users successfully accomplish each activity.

1. **Organizing and Scoping.** The organizing and scoping activity establishes the purpose, viewpoint, and context for the ontology development project, and assigns roles to the team members.

2. **Data Collection.** During data collection, raw data needed for ontology development is acquired.

3. **Data Analysis.** Data analysis involves analyzing the data to facilitate ontology extraction.

4. **Initial Ontology Development.** The initial ontology development activity develops a preliminary ontology from the data gathered.

5. **Ontology Refinement and Validation.** Ontology refinement and validation involves refining and validating the ontology to complete the development process.

232

Supporting this general process are IDEF5's ontology languages. There are two such languages: the IDEF5 *schematic language* and the IDEF5 *elaboration language*. The schematic language is a graphical language, specifically tailored to enable domain experts to express the most common forms of ontological information (see Figure A-24). This enables average users both to input the basic information needed for a first-cut ontology and to augment or revise existing ontologies with new information. The other language is the IDEF5 elaboration language, a structure textual language that allows detailed characterization of the elements in the ontology.

The central primitive symbols of the IDEF5 Schematic Language are shown in Figure A-24.
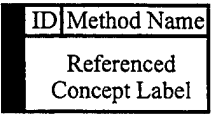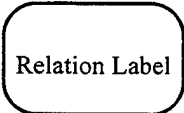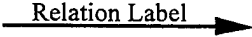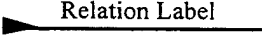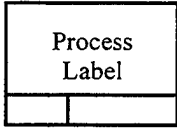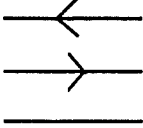
Kind Symbols; Individual Symbols; Referents	Relation Symbols; State Transition Symbols	Process Symbols; Connecting Symbols; Junctions
Kind Symbols Kind Label Individual Symbols Individual Label Referents ID Method Name Referenced Concept Label	*n*-Place First-Order Relation Symbols Relation Label Alternative Two-Place First-order Relation Symbols Relation Label Two-Place Second-Order Relation Symbols Relation Label State Transition Symbols Weak Transition Arrow Strong Transition Arrow Instantaneous Transition Marker Δ	Process Symbols Process Label Connecting Symbols Junctions X & O

Figure A-24

Basic IDEF5 Schematic Language Symbols

Various diagram types, or *schematics*, can be constructed in the IDEF5 Schematic Language. The purpose of these schematics, like that of any representation, is to represent information visually. Thus, semantic rules must be provided for interpreting every possible schematic. These rules are provided by outlining the rules for interpreting the most basic constructs of the language, then applying them recursively to more complex constructs.

However, the character of the semantics for the Schematic Language differs from the character of the semantics for other graphical languages. Specifically, each basic schematic is provided only with a *default* semantics that can be overridden in the Elaboration Language. The chief purpose of the Schematic Language is to serve as an *aid* for the construction of ontologies; the schematics themselves are not the primary representational medium for storing them. That task falls to the Elaboration Language. The Schematic Language is, however, useful for constructing *first-cut* ontologies in which the central concern is to record, in a rough way, the basic elements that exist in a domain, their characteristic properties, and the salient relations that can be obtained among objects of those kinds and among the kinds themselves. Consequently, the basic constructs of the Schematic Language are designed specifically to capture this type of information.[42]

During IDEF5's development, it was found that certain relations predominate when people express their knowledge about a domain. Because of their prominence and importance, these relations are included explicitly in the IDEF5 language. Four specialized schematic types derived from the basic IDEF5 Schematic Language are presented below to illustrate how IDEF5 can be used to capture ontology information directly in a form that is intuitive to the domain expert. The schematics presented include Classification Schematics, Composition Schematics, Relation Schematics, and Object State Schematics.

Classification Schematics. Among the more common structuring mechanisms used by humans to organize knowledge are taxonomy diagrams (Brachman, 1983). Domain experts engaged in knowledge acquisition often make statements such as **A is a B, A is a type of B**, or **A is a kind of B**. The cognitive activity involved in organizing knowledge in this fashion is called *classification*. There are several identifiable varieties of classification. Two particularly prominent types of classification are *description subsumption* and *natural kind classification*. In description subsumption, the defining properties of the "top-level" kind **K** in the classification, as well as those of all its subkinds, constitute rigorous necessary and sufficient conditions for membership in those kinds. Additionally, the defining properties of all the subkinds are "subsumed" by the defining properties of **K** in the sense that the defining properties of each kind entail the defining properties of **K**; the defining properties of **K** constitute a more general concept.

_____

[42] Refer to Knowledge Based Systems, Inc. (1994) for further information about the IDEF5 Elaboration Language.

Conversely, natural kind classification does not assume there are rigorously identifiable necessary and sufficient conditions for membership in the top-level kind K. Nonetheless, there are some underlying structural properties of its instances that, when specialized in various ways, yield the subkinds of K. The best examples of such classification schemes are, of course, genuine natural kinds such as **metal, feline**, and so forth; however, the idea can be extended to artifactual kinds like **automobile** and **NC machine**. These two types of classification are illustrated in Figure A-25.

Description
Subsumption

Natural Kind
Classification
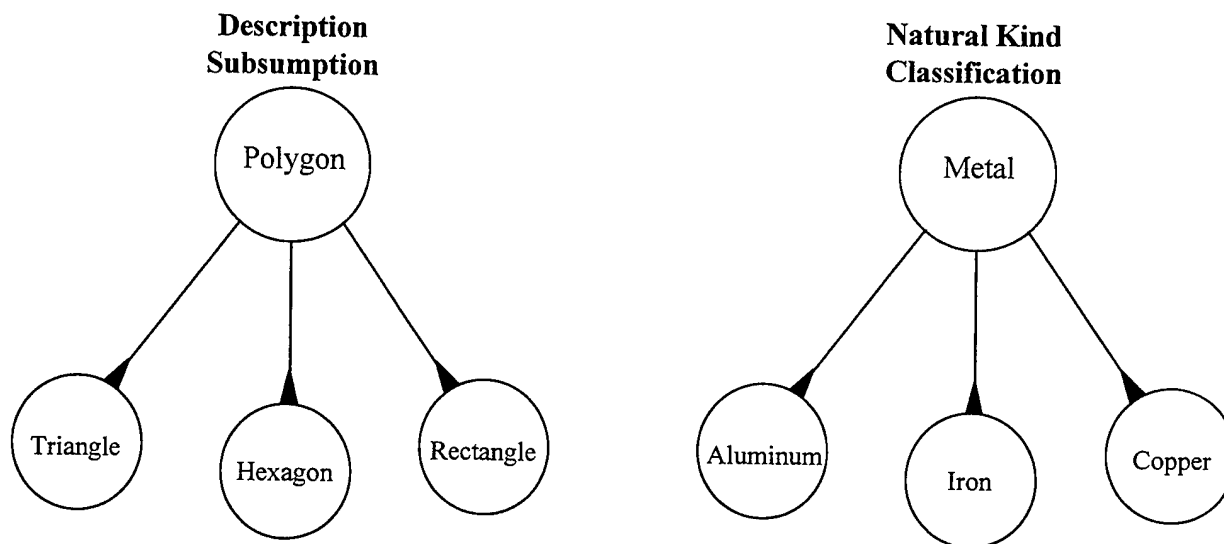
Figure A-25

Different Types of Classification

Clearly, with its central notion of a kind, a natural application for the IDEF5 schematic language is the development of taxonomy diagrams, or *classification schematics*.

Classification is typically much more detailed than the examples suggest. Most classification schemes will involve several levels of specialized subkinds "below" general kinds in the scheme.[43] To illustrate, it is essential in project planning to categorize the kinds of *resources* that will be needed for the project's success. Informally, a resource can be defined as an object that is consumed, used, or required to perform activities and tasks. Therefore, resources play an enabling role in processes. Classification diagrams provide a natural way of categorizing necessary resources. See, for example, Figure A-26. Note that second-order relation symbols with no attached labels default to the subkind relation.

---

[43] Such schemes are often called "is-a hierarchies," but the use of "is-a" is strongly discouraged in IDEF5. Either the *subkind-of* relation or the *instance-of* relation should be used instead, depending on the intended meaning.
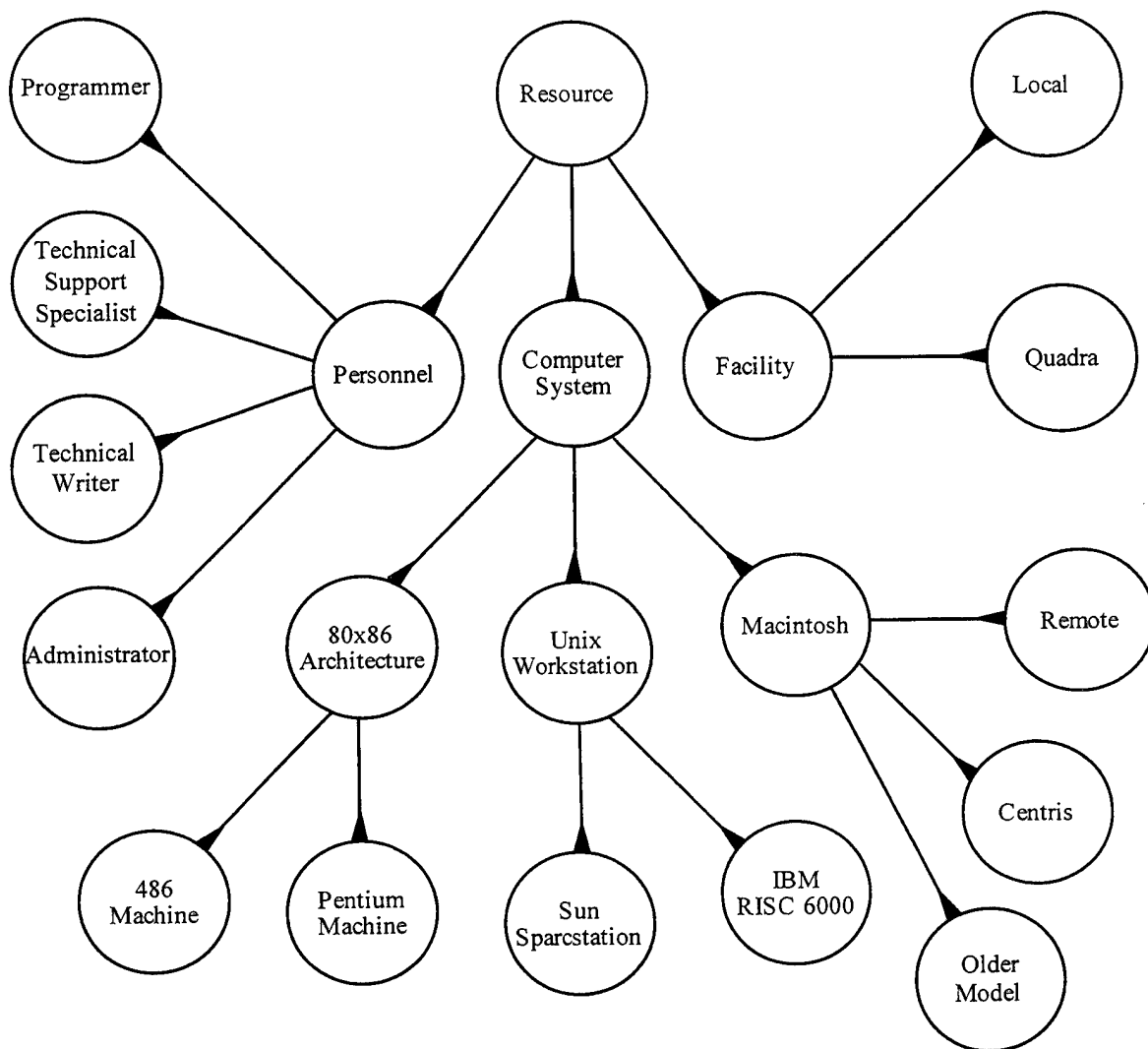
Figure A-26

Classification of Resources

As with complex composition schematics, however, it is often useful to hide some detail in a classification schematic. Thus, in some contexts (e.g., those in which facilities and personnel need to be highlighted), information about computer systems might not need to be explicit. As with composition schematics, hidden information will be indicated by a double circle, annotated in this case with an upper case "C" (for *classification*) at the top of the circle as shown in Figure A-27. Thus, one might hide that information and add information about facilities to obtain the following schematic.
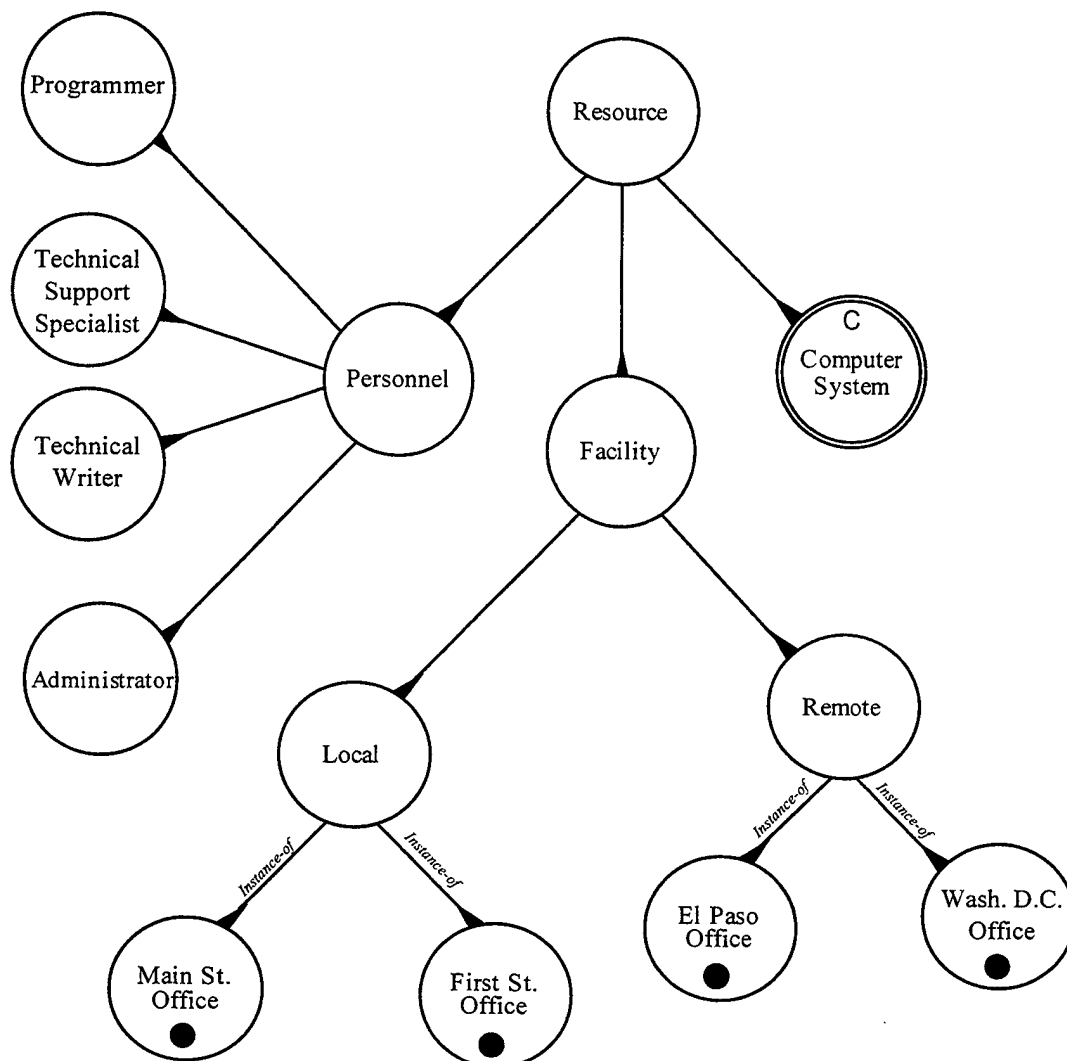
Figure A-27

Classification of Resources with Hidden Information

Composition Schematics. Because the *part-of* relation is so common in design, engineering, and manufacturing ontologies, the "part-of" label and its associated axioms are included in the IDEF5 languages. In particular, this capability enables users to express facts about the composition of a given kind of object. In general, this is achieved by means of schematics of the form illustrated in Figure A-28.

For example, one might wish to represent the component structure for a certain kind of ballpoint pen, as in Figure A-29.

237
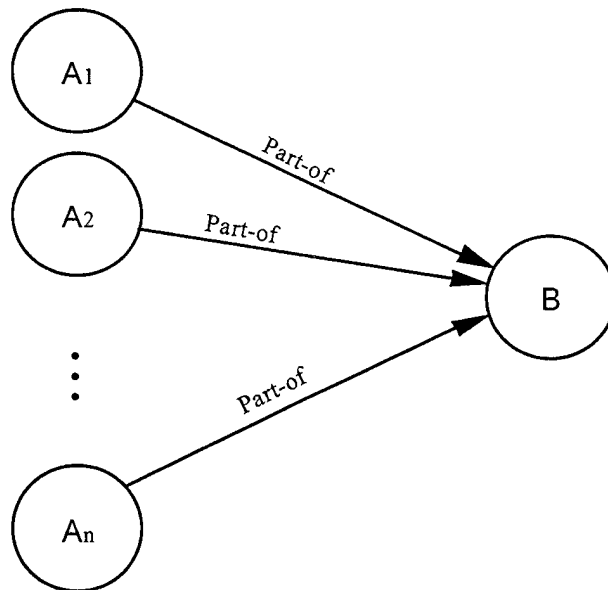
Figure A-28

Composition Schematic
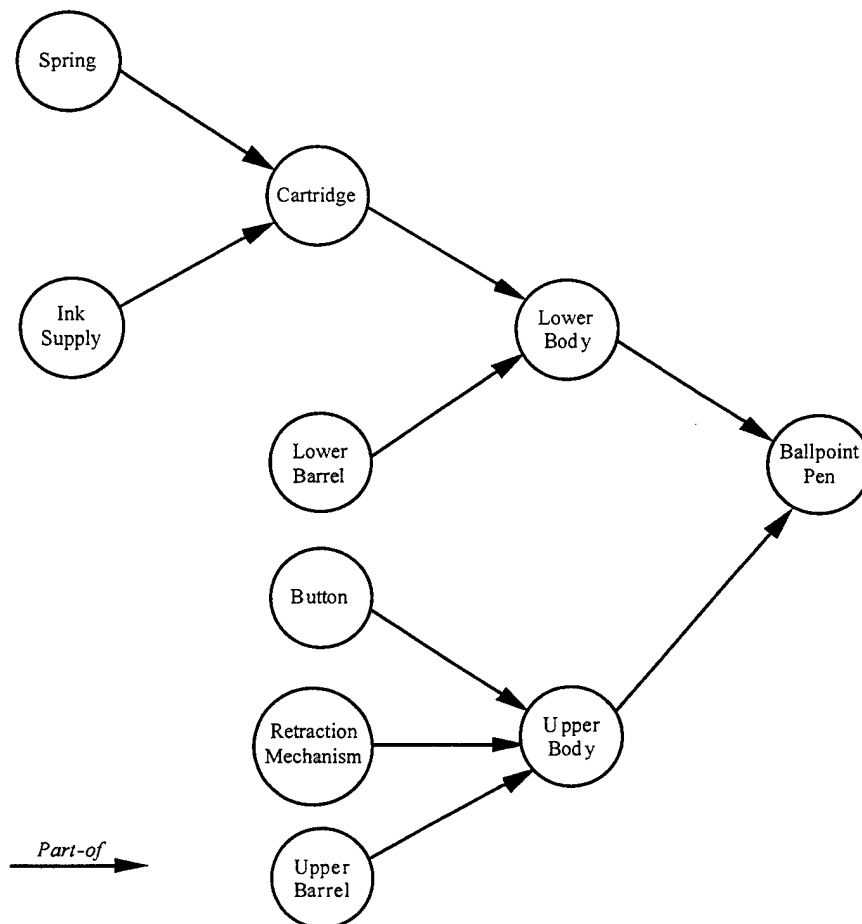
Figure A-29

Composition Schematic for the Kind Ballpoint Pen

The schematic in Figure A-29 shows that a ballpoint pen in the domain in question has both an upper body and a lower body, and that the former consists of a button, a retraction mechanism, and an upper barrel while the latter consists of a lower barrel and a cartridge, which in turn consists of a spring and an ink supply.[44]

As Figure A-29 illustrates, composition schematics can be quite detailed. Such detail can cause a great deal of clutter in an IDEF5 diagram. For instance, in addition to describing the component structure of the kind *ballpoint pen*, one might also want to talk about many of the other relations it and its instances are involved in — for example, that the pens can be made in Sequim, Washington; that fountain pens generally cost more than ballpoint pens; that *ballpoint pen* is a subkind of *pen*, and so on. Hence, in many contexts, the component structure of the kind might be irrelevant; in such cases, it would be useful to be able to hide that information. That such information is being hidden is indicated on a diagram by using a double circle to represent the kind (instead of a standard single circle), along with an upper case "P" (for *part-of*) at the top of the circle to distinguish the kind of information that is being hidden, as illustrated in Figure A-30.
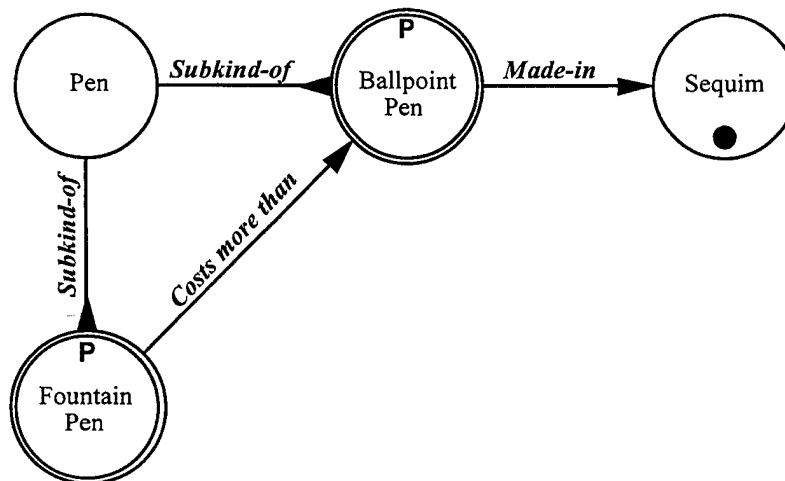
Figure A-30

Hiding Composition Information

---

[44] The default semantics for first-order relations, like the *part-of* relation, is that it is possible or permissible for individuals represented to bear the specified relation to one another. A stronger interpretation can be imposed via the Elaboration Language. Specifically, in the case of a kind B whose instances have three parts of kinds A1, A2, and A3, one would add the elaboration language statement (forall ?x (-> (B ?x) (exists (?y1 ?y2 ?y3)(and (A1 ?y1) (A2 ?y2) (A3 ?y3) (part-of ?y1 ?x) (part-of ?y2 ?x) (part-of ?y3 ?x))))).

Relation Schematics. The IDEF5 Relation Schematics allows ontology developers to visualize and understand relations among kinds in a domain. Consider the source statement: **a manpower planner develops a manpower plan in which resources are allocated to perform the required activities.** Based on this source statement, the ontology development team might create the relation schematic shown in Figure A-31 or, alternatively, Figure A-32.
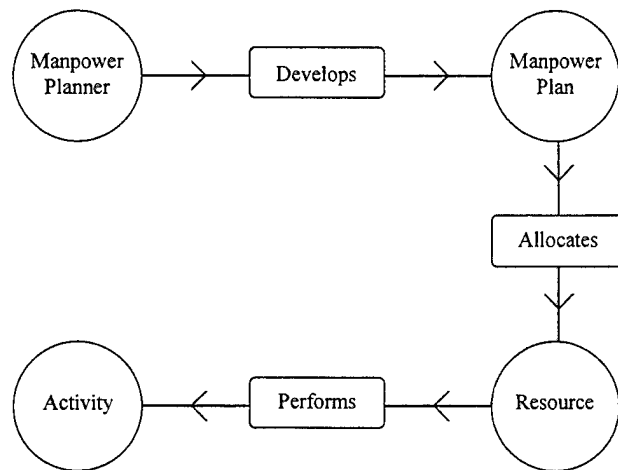
Figure A-31

First-Order Schematic
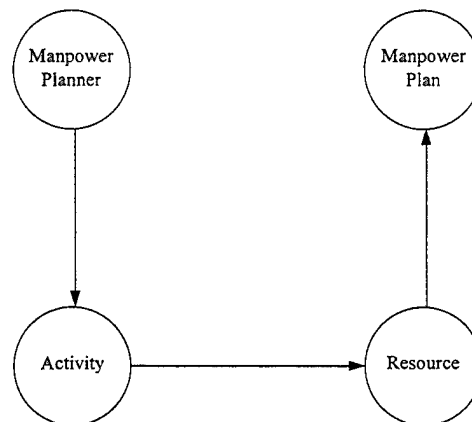
Figure A-32

Alternative Syntax for the Schematic in Figure A-49

IDEF5 Relation Schematics can also be used to capture and display relations between first-order relations. The motivation for developing this capability is that people often describe and discover new concepts in terms of existing concepts. This means of creating and defining new concepts is consistent with Ausubel's theory of learning, wherein learning often occurs by subsuming new information under more general, more inclusive concepts (Novak & Gowin, 1984; Sarris, 1992). Based on this hypothesis, a natural way to describe a new (or poorly understood) relation is to connect it to a relation that is already well understood and, more generally, to categorize

its place in a "conceptual space" of other relations. The IDEF5 relation library provides a baseline reference to help users discover and characterize relations.

The basic syntax used to display this kind of higher-order relation is illustrated in Figure A-33.
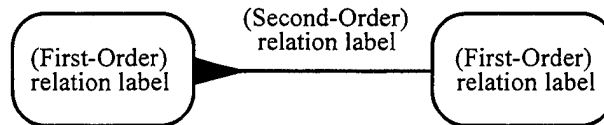
Figure A-33
General Form of a Basic Relation Schematic

A second-order relation symbol is needed because kinds and first-order relations are of the same logical type. Kinds are properties of individuals; first-order relations are associations between individuals. Thus, relations between first-order relations are of the same logical type as relations between kinds: both relate entities that hold (or do not hold) with respect to individuals and, hence, are second-order relations.

This kind of relation schematic is used to facilitate conceptual analysis involving relations (both relations between kinds and relations between relations). Consider the ontology for an engineering Bill of Materials (BOM) of an automobile manufacturing company. The *part-of* relation plays an important structural role in the BOM. The data acquired during the collection process might contain the following statements about the BOM:

- An automatic transmission is a variant of the transmission.

- A manual transmission is a variant of the transmission.

- A radio is an option of the car.

A *variant* of a product is an essential characteristic of the product that is often determined by customer choice (Anupindi, 1992). The customer picks one of several possible variants. In this example, automobile customers choose between automatic and manual transmissions. Notice that *having a transmission system* is an essential property of the car (for most contemporary automakers). The particular variety of transmission is a choice exercised by the customer.

An *option* is a feature of a product that the customer chooses. Options are different from variants in that options can be eliminated completely from a product, whereas variants are required characteristics. In the example of the car, a radio is not essential to the function of the car; thus, it can be optionally excluded.

Based on an analysis of the source statements, the IDEF5 developer hypothesizes the existence of two relations and selects the relation names *variant-of* and *option-of*. These relations are mapped graphically on a relation schematic in the manner shown in Figure A-34.

241
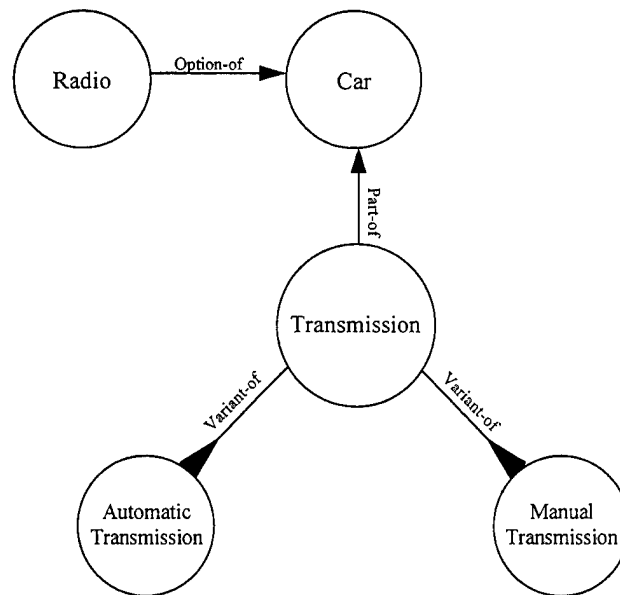
Figure A-34

Bill of Material Relation Schematic

At this stage, an IDEF5 developer would browse the IDEF5 Relation Library and may notice that the *variant-of* and *option-of* relations are conceptually similar to some relations. Specifically, the developer may realize that the *option-of* relation is a specialization of the *part-of* relation. These insights can be represented by the second-order relation *specialization-of* in the relation schematics (Figure A-35).
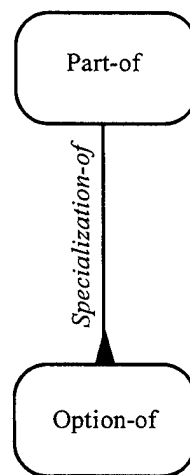
Figure A-35

Relation Schematics Involving the *Specialization-of* Relation

The higher-order *specialization-of* relation is used to assert the generalization-specialization relation between the indicated first-order relations. Further reflection leads the IDEF5 developers to draw a more detailed relation schematic that places the *option-of* relation in a relation taxonomy diagram (i.e., a special type of relation

schematic that shows a hierarchy of relations that are associated by generalization-specialization relationships). The taxonomy in question is exhibited in Figure A-36.[45]
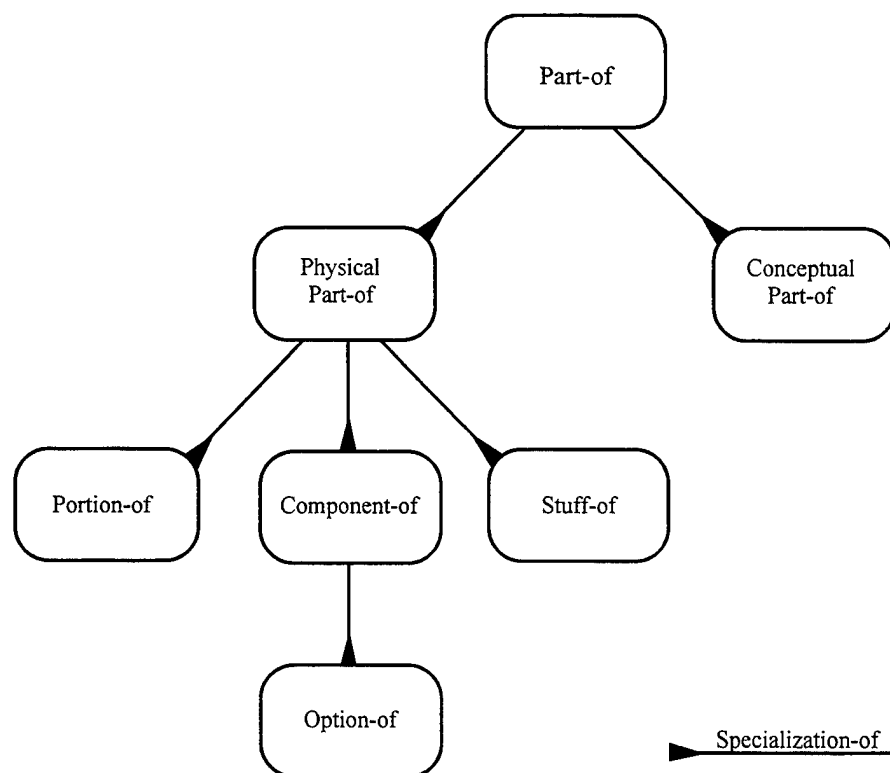
Figure A-36

A Partial Relation Taxonomy of the *Part-of* Relation

Object State Schematics. There is no clean division between information about kinds and states and information about processes. This subsection describes how the IDEF5 schematic language enables modelers to express fairly detailed *object-centered* process information (i.e., information about kinds of objects and the various states they can be in relative to certain processes). Diagrams built from these constructs are known as *Object-State Schematics.*

Two types of changes can be observed in the objects undergoing processes: change in kind and change in state. There is no *formal* difference between these two types of change: objects of a given kind **K** that are in a certain state can simply be regarded as constituting a subkind of **K**. For formal purposes, for example, **warm water** can be regarded as a subkind of **water**. However, it is useful to distinguish the two in the schematic language to

---

[45] A more complete taxonomy diagram of the meronymic (part-of) relations is given in the IDEF5 Relation Library discussed in Knowledge Based Systems, Inc. (1994).

indicate explicitly the kind of thing that is in a certain state. This is done using colon notation (e.g., **kind:state**).
For example, warm water will be indicated by the label **water:warm**, frozen water by **water:frozen**, and so on.
The notation is illustrated in Figure A-37.
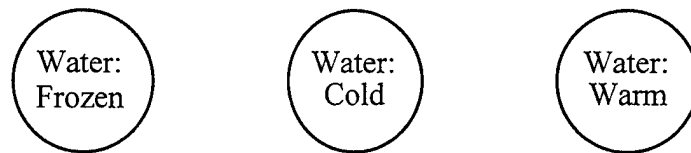
Figure A-37

Kinds and States

In this context, the *subkind-of* relation is best thought of as a *state-of* relation, as illustrated in the
classification diagram in Figure A-38.
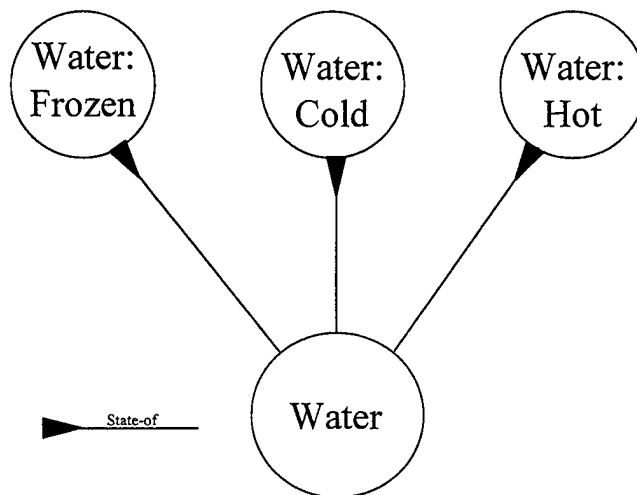
Figure A-38

Schematic Depicting States of Water

Indicating how objects change either kind or state within processes requires an entirely new class of
construct. This need is addressed by the Object-State Schematics.

The first and most basic construct is the simple *transition* link shown in Figure A-39. The presence of the
open circle distinguishes an object state transition link from a general relation arrow.
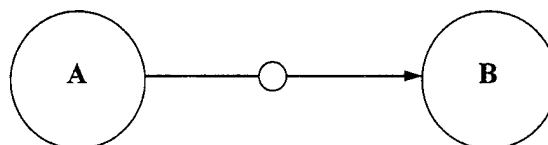
Figure A-39

Basic State Transition Schematic

244

The circles labeled A and B in these links indicate two kinds, and everything said previously about kinds and kind symbols still applies. However, in making general characterizations about the manner objects are transformed through a process, it is natural to classify those objects in terms of their *states* at various stages of the process. Hence, the kind symbols in a state transition schematic typically will indicate a kind *in* a particular state, such as **dry wood**, **warm water**, **unreworked part**, and so forth. To emphasize this role of the kind symbols, their meanings will often be referred to as *object states* or, simply, *states*.

Intuitively, an object state transition link indicates there is an allowable transition such that an object in a given state **A** may be modified, transformed, or consumed so as to yield an object (possibly the same object) in a different state. Figure A-39 depicts the situation in which a certain type of transition from **A** to **B** is observed, but no knowledge or desire to specify the process(es) involved in the transition exists.

It is important to note the distinction between the characterization of an object in a given state and the conditions or rules that govern how the object transitions to and from that state. In the IDEF3 process description capture method, conditions for entering and leaving a state are called, respectively, *entry* and *exit* conditions. The IDEF5 Elaboration Language can be used to specify relevant entry and exit conditions for a given state.

Additional information about the process(es) involved in a state transition may be displayed in IDEF5, as shown in Figures A-40 and A-41.
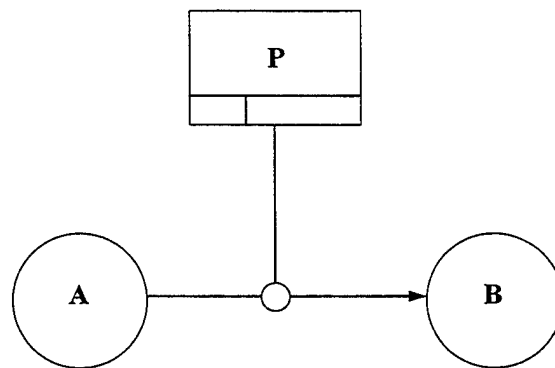
Figure A-40

Schematic for Object State Transition within a Process

The state transition schematic displayed in Figure A-41 indicates that during any occurrence of the process **P**, an object **a** in state **A** will transition to a possibly different object **b** in state **B**. However, an identified transition might occur not *within* a process but *between* the end of a given process **P** and the start of some process **Q** (see Figure A-42). A special case of this situation may occur where two contiguous processes **P** and **Q** meet.
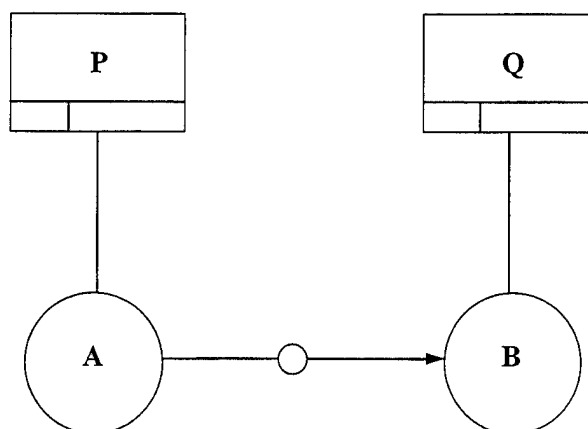
Figure A-41

Schematic for Object State Transition between Processes

The semantics of an object state transition link do not definitively answer the question of whether the object in state **A** at the beginning of a transition is *identical*[46] to the object in state **B** after the transition (e.g., when an unpainted object becomes a painted object) or *distinct* (e.g., when a piece of wood is transformed into a pile of ashes by a furnace). The basic state transition schematic should be used in any of the following three cases: the objects at the beginning and end of the process are distinct; it is not known whether they are distinct; or it does not matter if they are distinct. On the other hand, if a modeler desires to show explicitly that the object at the beginning of an instance of a state transition is identical with the object at the end, a *strong* state transition link should be used. This involves simply affixing a double tip on the arrow in a state transition link, as shown in Figure A-42.

These schematics do not require that the transition of the object from state **A** to state **B** be instantaneous. There may be an intervening period in the process during which an object in state **A** is being transformed into something in state **B** yet there is actually nothing in either state, as in the period during which a quantity of water is heated from 5°C (state **A**) to 100° (state **B**) (Figure A-43).

---

[46] Identity may be in terms of chemical structure, mass, physical form, function, and so forth. For example, grape juice becomes wine after undergoing the fermentation process. One might argue that the "stuff of" the kind **grape juice** is the same as that of the resulting kind **wine**. Other people having different attunements may perceive the two kinds as being entirely different based on, for example, chemical composition of the two kinds. It is therefore recommended that the assumed criteria for identity be established or characterized when there is possible ambiguity.
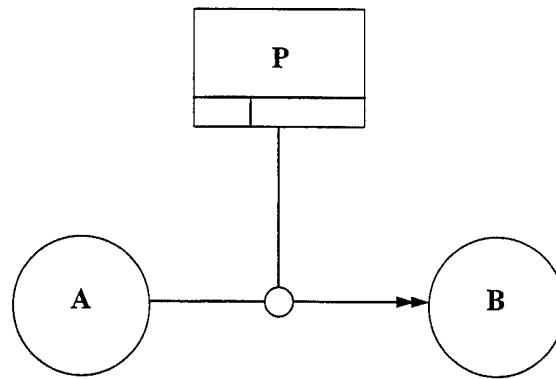
Figure A-42

Strong State Transition Schematic
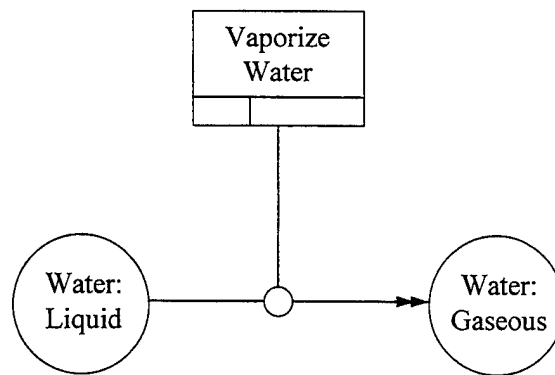
Figure A-43

Example of Strong State Transition Schematic

To explicitly represent instantaneous state transitions within a process (relative to a certain temporal granularity), the OS syntax allows a modeler to tag the small circle in an object state transition link with a Δ (Figure A-44), indicating that the transition occurs over a period smaller than the smallest time unit D recognized in the context being modeled.
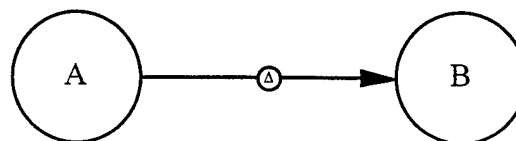
Figure A-44

Instantaneous State Transition Schematic

A double arrow tip, of course, may be added if a strong transition is desired. For instance, when liquid oxygen is exposed to atmosphere, it can be considered to transform to gaseous state instantaneously (Figure A-45).
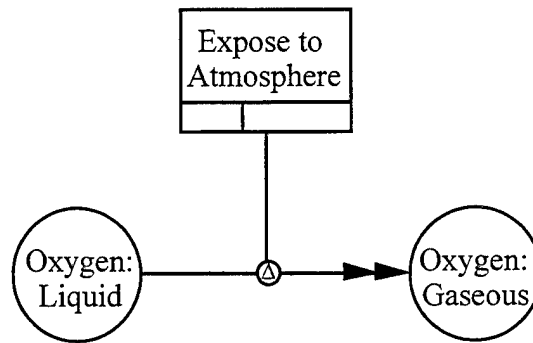
Figure A-45

Example of Instantaneous State Transition

However, the area of interest may be the processes which immediately precede and/or follow the transition rather than the instantaneous process which occurs upon transition. This situation is illustrated by the interval diagram in Figure A-46.
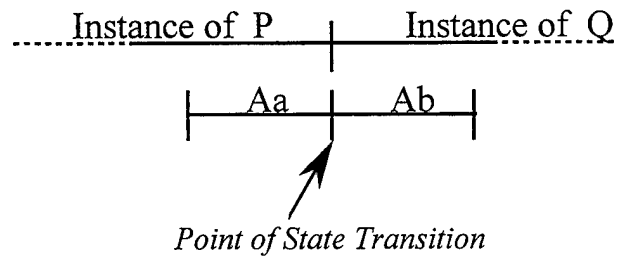
Figure A-46

Interval Diagram for Figure A-47

To express the content of Figure A-46 precisely, the construct in Figure A-47 is used. The transition arrow in such a schematic marks the division in time between an object in state **A** at the end of some process **P** and the transformation of that object to state **B** at the start of another process **Q**.
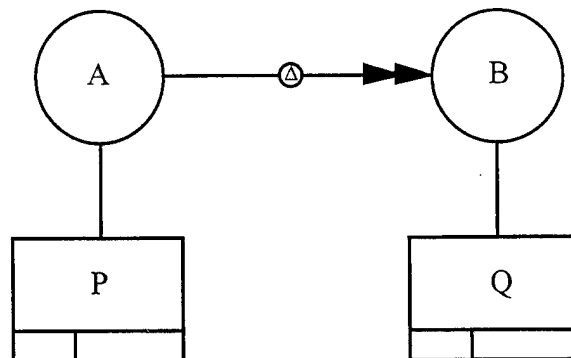
Figure A-47

Precise Expression of Figure A-46

To illustrate further, imagine a cutting tool that is driven forward across a workpiece until it activates a limit switch whereupon the cutter is switched off and retracted to its starting position (Figure A-48).
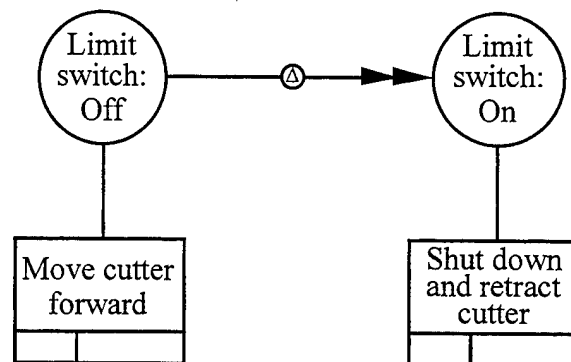
Figure A-48

Cutoff Switch Example for Figure A-47

The IDEF5 Relation Library. IDEF5 also includes an extensive relation library that constitutes a knowledge-rich repository of definitions and characterizations of commonly used relations. It provides a repository of formally defined and characterized relations that can be reused and customized. The motivation for this library developed from an analogy with software engineering. Often, in software development, the same kinds of routines are used in different programs by (in general) different programmers. In earlier times, great amounts of time and effort were lost because of the inability to reuse work. Recognition of this problem has led to the development of extensive *libraries* that contain frequently used routines which programmers can call straight into their programs. Such libraries have eliminated the need to duplicate the functionality of existing code. The development of ontologies will face the same sort of problem (and solution). The same or similar relations will likely appear in a number of different ontologies. By reusing IDEF5's library of relations, users can customize relations that have been defined in previously captured ontologies. The library can also serve as a reference for the different methods of defining and characterizing relations and illustrative examples of IDEF5 elaboration language use. All definitions and characterizing axioms in the library are written using the IDEF5 elaboration language. Finally, the library is extensible in that any relation that has been formally defined and characterized may be added to it.

The IDEF5 library relations are grouped into the following seven categories:

1. Classification Relations (including class inclusion relations).

2. Meronymic (i.e., part-of) Relations.

3. Temporal Relations.

4. Spatial Relations.

249

5. Influence Relations.

6. Dependency Relations.

7. Case Relations.

Figure A-49 illustrates the IDEF5 relations categorization.
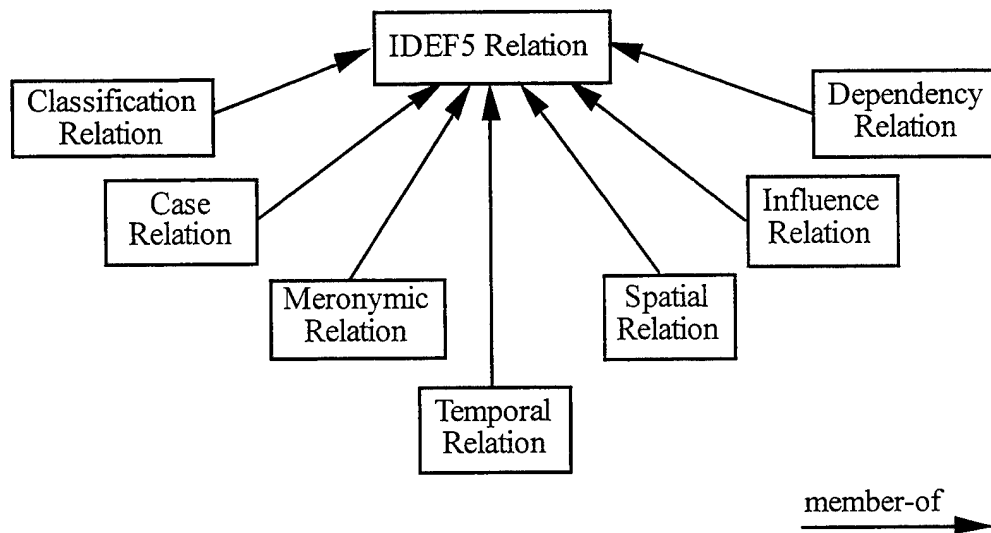
Figure A-49

Overview of the IDEF5 Library Relations

Summary. The nature of any domain is revealed through three elements: the vocabulary used to discuss the characteristic objects and processes comprised in the domain, rigorous definitions of the basic terms in that vocabulary, and characterization of the logical connections between those terms. An *ontology* is a domain vocabulary complete with a set of precise definitions, or *axioms*, that constrain the meanings of the terms in that vocabulary sufficiently to enable consistent interpretation of data that use that vocabulary. The IDEF5 method provides a structured technique by which a domain expert can effectively develop and maintain usable, accurate domain ontologies. The IDEF5 method is used to construct ontologies by capturing assertions about real-world objects, their properties, and their interrelationships in an intuitive and natural form.

A more detailed description of the IDEF5 Ontology Description Capture method is available in the IDEF5 Method Report (Mayer et al., 1994).

IDEF9 Business Constraint Discovery Method Overview

Policies, rules, conventions, procedures, contracts, agreements, regulations, and societal and physical laws define the structure of an enterprise. These items are the mechanisms for forging relationships between people,

250

information, material, and machines to make a system. This report refers to these items collectively as constraints. When viewed as a machine, an enterprise can be seen as having constraints which form the architecture and the programming language that define the behavior of that machine.

The IDEF9 Business Constraint Discovery method was designed to assist in the discovery and analysis of constraints in a business system. A primary motivation driving the development of IDEF9 was the observation that the collection of constraints that forge an enterprise system is generally poorly defined. That is, the knowledge of what constraints exists and how they interact is incomplete, disjointed, distributed, and often completely unknown. This situation is not necessarily alarming. Just as living organisms need not be aware of the genetic or autonomous constraints that govern certain behaviors, an organization can (and usually does) perform well without a complete understanding of the "glue" that structures its system. However, if a business desires to modify its structure to improve its performance or to adapt to changes in a predictable manner, knowledge of these constraints can be vitally important.

Organizations benefit from IDEF9 because it facilitates the discovery and mapping of the relevant constraints in an organizational system. Once these constraints have been cataloged, they can be examined systematically and, if necessary, tuned or replaced to improve the performance of the system (Table A-2).

Table A-2. Some Benefits of Constraint Discovery

Constraint-Related Problem	Benefit of Constraint Discovery
The cost of maintaining a constraint exceeds the value of the constraint.	Enable decision-makers to identify and eliminate constraints that exceed the value they provide.
A constraint no longer support organizational goals.	Enable decision-makers to identify and eliminate outdated or unnecessary constraints.
The constraint causes unintended or undesirable effects.	Enable decision-makers to reengineer or eliminate constraints that produce unintended or undesirable effects.
The agent or system (mechanism) responsible for maintaining the constraint fails to consistently or correctly enforce the constraint.	Enable system developers to identify and remedy system designs that fail to appropriately enforce constraints.
What was believed or intended to be a constraint lacks any explicit maintenance or enforcement mechanism.	Enable decision-makers to identify missing systems needed to maintain both precautionary and enabling constraints.

Constraints often serve a dual role as both the glue and the rationale for a system. That is, the collection of relevant constraints often constitutes the description of why the system behaves as it does. From this perspective, IDEF9 provides a reverse engineering tool for the business engineer. It can help the user discover the "logic"

behind the design of an existing system and also provides a mechanism for specifying the logic of a "To-Be" system.

The ability to catalog business constraints can assist decision-makers in designing and prioritizing constraints relative to organizational goals. Knowledge of interrelationships among constraints also enables more reliable performance predictions and change impact assessments. By identifying and eliminating outdated or unnecessary constraints, costs are eliminated, improvements in performance can be realized (e.g., schedule and quality gains), and freed resources can be used to leverage new opportunities. Often, these benefits can be realized without any additional investment in automation or information systems.

Somewhat less obvious is the fact that knowledge of constraints can yield new sources of information and expose misinformation. For example, knowledge of the constraint that trees add one ring to their circumference each year, when coupled with a knowledge of the number of rings in a given tree, yields new information — the age of the tree. Similarly, knowledge of business constraints can be used:

- to discover information about the health and productivity of the business,

- to determine how quickly and cost effectively products can be produced,

- to estimate what it takes competitors to produce their products, and

- to identify where changes can be made to achieve competitive advantage.

Constraints can be broadly classified as either *enabling* or *limiting* within a given context (Figure A-50). Although the term *constraint* often evokes images of negative influence or control, constraints serve the vital and enabling role of establishing the system. Tolerances between mating parts, for example, establish the constraints required to ensure correct fit. A company's fiscal management policies and accounting procedures not only stabilize the health of the business but work to prevent unmanageable debt, fraud, and waste. Both the limiting and enabling aspects of constraints are evidenced in alternative definitions for a constraint in the literature. For example, in *Theory of Constraints*, Eliyahu Goldratt defines a constraint as "anything that limits a system from achieving higher performance versus its goal" (Goldratt, 1990). Other definitions state that constraints are "the rules, requirements, relations, conventions, and principles that define the context of designing" (Gross, Ervin, Anderson, & Fleisher, 1987), "specifications, requirements, needs, performance measures, and objectives" (Smith & Browne, 1993), and "a characteristic of the environment, or of the artifact as currently conceived, [that] rules out or against potential settings of design variables" (Smith & Browne, 1993). In other words, whenever the term is used, there is an implicit thought that a constraint can be an enabling or limiting factor in design or on performance.
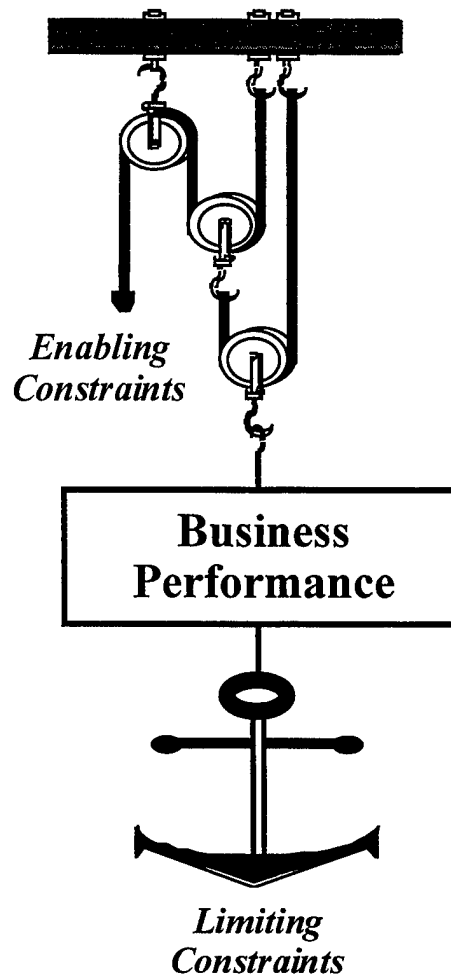
Figure A-50

Constraints: Enabling or Limiting

Limiting constraints are, of course, the most obvious because they manifest themselves through problematic symptoms. Bottlenecks, excessive costs, low quality, long development lead times, waste, and inefficiency are all symptoms of limiting constraints. Symptoms are one form of *evidence* that constraints exist within the system.

Symptoms are one form of *evidence* that constraints exist within the system. *Evidence* is an indication, sign, or manifestation that supports the existence of a constraint in a given context. Evidence of both enabling and limiting constraints can take many forms, including symptoms (observable evidence of a system failing to meet goals), operating instructions, procedure manuals, employee handbooks, regulations, specifications, policy manuals, project files, design models, and so forth. These are not the constraints themselves but indications of possible constraints. Policy statements in a policy manual, for example, would provide evidence supporting the proposition that there are constraints whose descriptions are found in the policy manual. However, if the policies are not maintained or enforced, no constraint exists.

The need to maintain or enforce a relation so that it qualifies as a constraint implies the need for some agent or system; that is, the existence of a constraint implies the existence of a system that maintains or enforces the relationship. However, a constraint must not be confused with the system that maintains the constraint. A constraint is simply a special kind of association between a relationship that is maintained and the system that maintains that relationship.

Because constraints are enforced or maintained through the use of a system, they exist at a cost. In the natural world, balance sheets and cash flows are not used to reflect the cost of constraints. Instead, the maintenance of natural constraints is reflected in terms of energy expended or transformed. However, business constraints are maintained by business systems, and operating business systems costs time and money.

Using IDEF9 to Discover Business Constraints. Constraint discovery is an evolutionary process through which candidate constraints are identified, validated, and refined. In general, when using IDEF9 to discover and document constraints, the following six steps are applied recursively:

1.   Collect — Acquire observations and sources of evidence for constraints.

2.   Classify — Isolate and distinguish contexts, objects, object types, properties, and relations.

3.   Hypothesize — Postulate candidate constraints from the data and evidence gathered.

4.   Substantiate — Generate or collect examples to determine which candidate constraints merit promotion to the status of a constraint.

5.   Challenge — Involve domain experts in testing the validity of analysts' conclusions.

6.   Refine — Filter, improve, adjust, and add detail to constraint characterizations.

Graphical Language Developments

To support these basic steps, seven candidate schematics were identified and prototyped:

1.   Context schematic

2.   Constraint resource schematic

3.   Constraint relationship schematic

4.   Constraint effects schematic

5.   Goal schematic

6.   Goal relationship schematic

7.   Symptom schematic

Each candidate schematic is described below.

Context Schematic. The Context Schematic is designed to help users establish and incrementally refine the scope of a constraint discovery effort, display the constraints that hold in a given situation, and rapidly identify shared constraints among distinguished contexts. Figure A-51 displays the candidate syntax for this schematic.
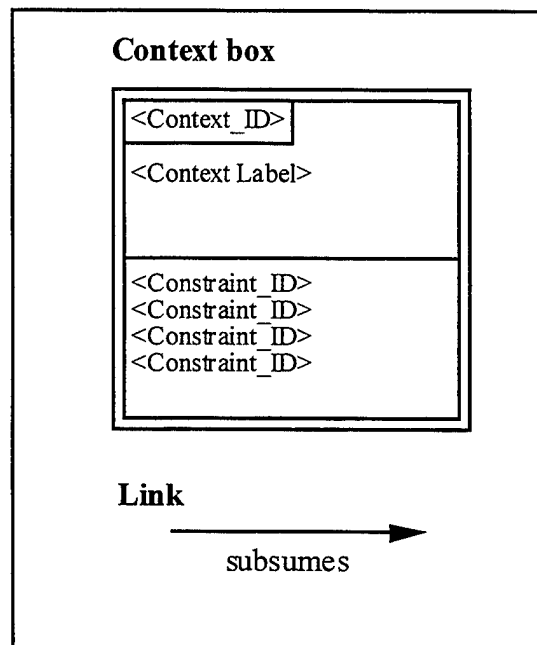
Figure A-51

Candidate Syntax for the Context Schematic

The example in Figure A-52 illustrates five distinguished contexts (V, W, X, Y, and Z). Each context is distinguished by a unique identifier and provided with a descriptive label. A more detailed description of the context would be included in an elaboration form supporting the schematic. Although the central focus of this schematic is the context, graphical depiction of the anatomy of the context (i.e., the facts, objects, and relations that collectively define the context) is unnecessary and possibly burdensome. The primary purpose for the schematic is to organize constraints in terms of the situations in which they hold. Thus, the only information about the context that must be displayed is that needed to distinguish one context from another. As indicated by the example above, constraints C1 and C2 are listed in the context box identified as X. The arrow leading from context X to context Y shows that context Y is a subcontext of context X. That is, all that was true in X is also true in Y — with possibly

more constraints. Contexts can be merged to form new contexts, as is shown by context W, a combination of context X and context Z. Notably, this process is strictly additive; that is, new constraints are always added to the contexts as you move down the schematic. You cannot remove constraints as you move down the schematic.
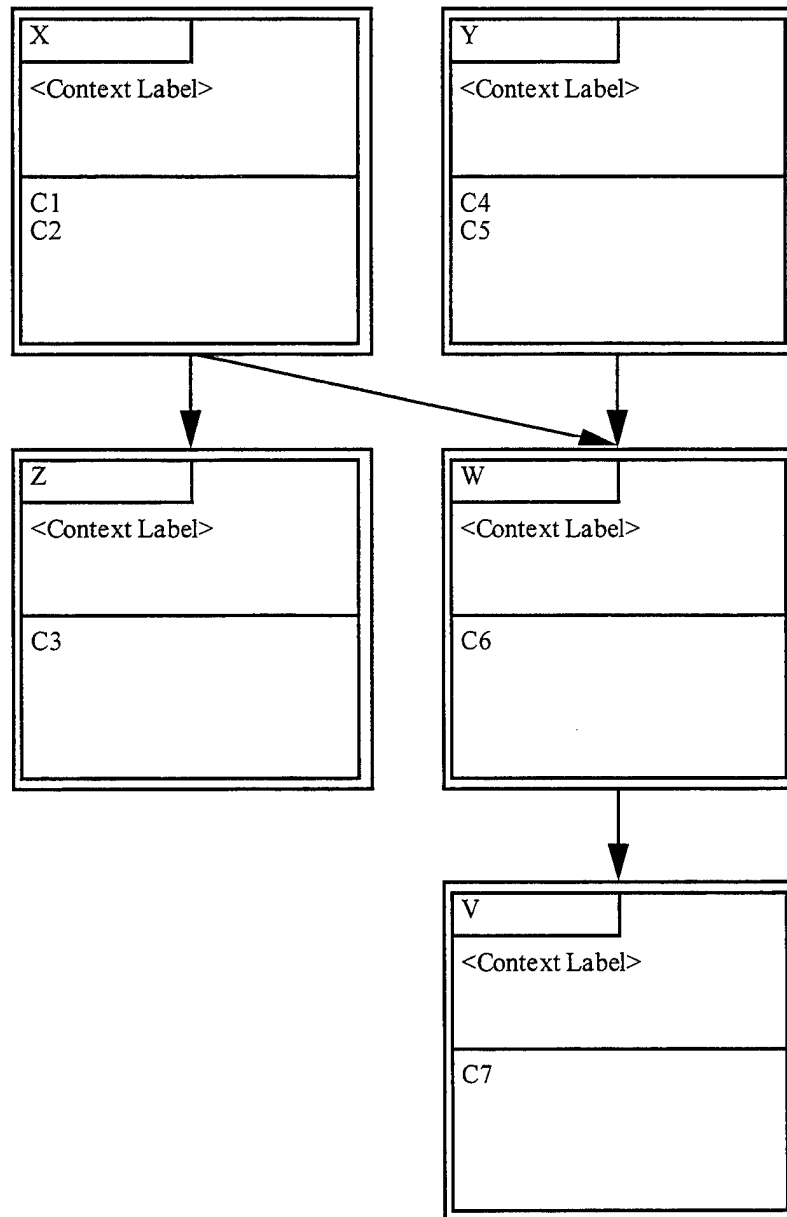
Figure A-52

Example Context Schematic

Constraint Resource Schematic. The syntax for the Constraint Resource Schematic (Figure A-53) is designed to allow users to display the system(s) or object(s) that maintain or enforce the constraint.
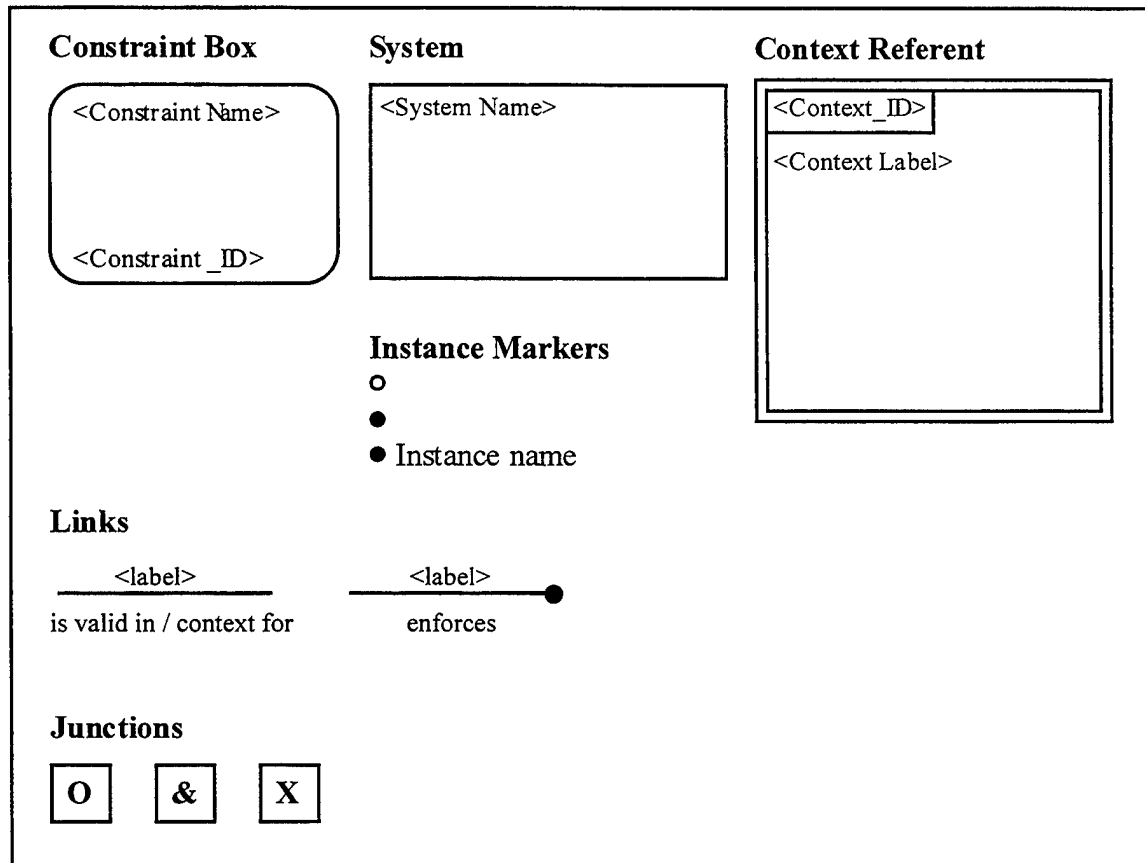
Figure A-53

Candidate Syntax for the Constraint Resource Schematic

Figure A-54 shows an example using this syntax. In this example, the Comptroller (Harriet Smith) and the Board of Directors together maintain the "Dividends Distribution" constraint during end-of-year close-out. The Board of Directors is responsible for representing the shareholders in ensuring that the allocation of end-of-year profits toward stock value and dividends maintain an acceptable level of Return on Investment (ROI). Acting on the majority vote of the Board of Directors, the Comptroller invests profits and/or distributes dividends to stockholders.

257
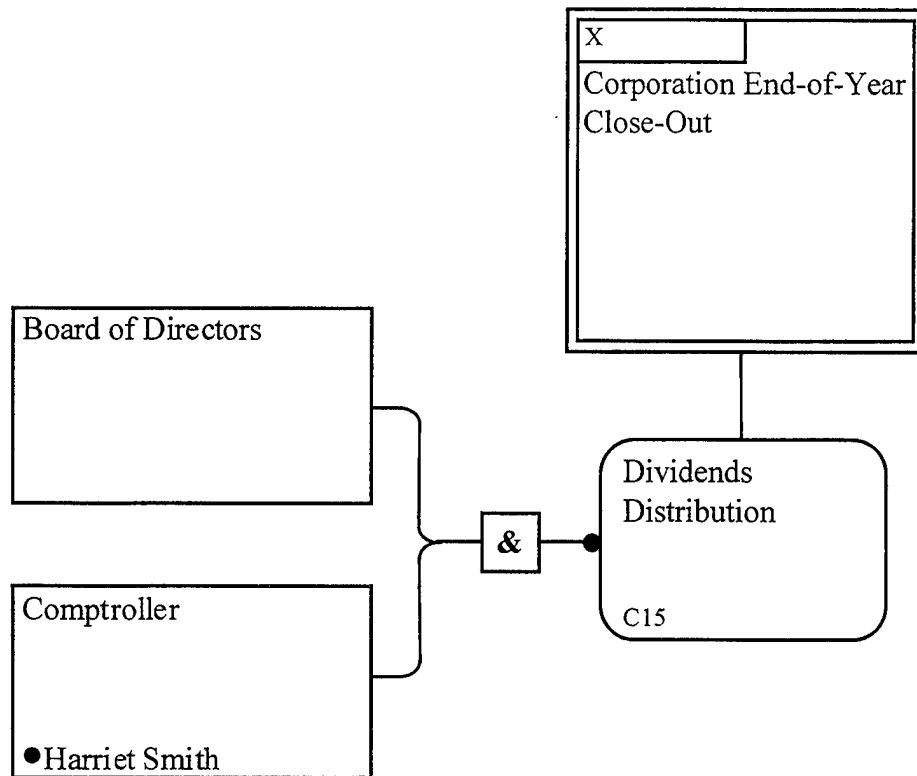
Figure A-54

Example Constraint Resource Schematic

Constraint Relationship Schematic. The Constraint Relationship Schematic is used to display existential dependency, part-of, and user-defined relationships among constraints. The proposed syntax for this schematic is displayed in Figure A-55.

These graphical elements can be combined to capture important relationships among constraints. For instance, the example illustrated in Figure A-56 demonstrates that if students in the Master's degree program did not have to submit a thesis, they would also not have to pay a binding fee.

Constraint Effects Schematic. The Constraint Effects Schematic displays the object(s) and system(s) affected by a constraint. Figure A-57 displays the candidate syntax for this schematic.

Constraint Box

<Constraint Name>

<Constraint_ID>

Context Referent

<Context_ID>

<Context Label>

Links

implies

is valid in/context for

user-defined

depends-on-existentially

is-part-of
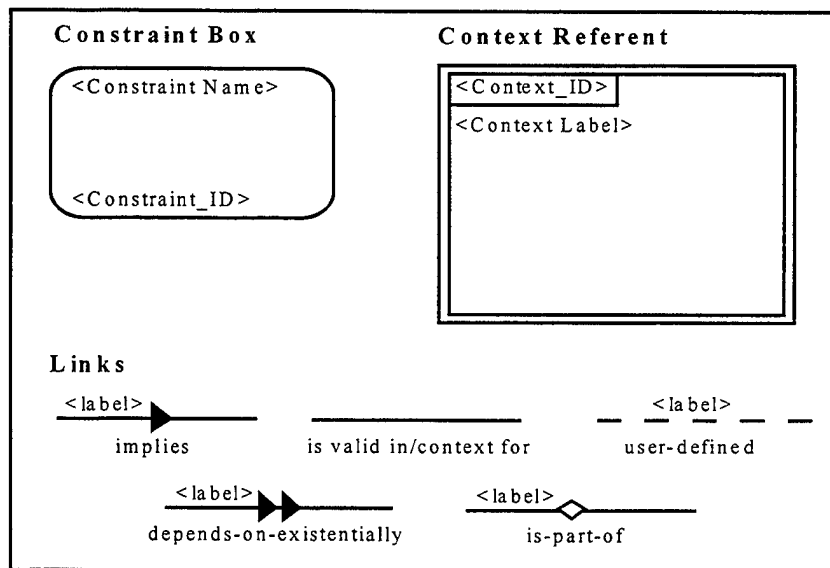
Figure A-55

Candidate Syntax for the Constraint Relationship Schematic

Y

State University Masters of Science (MS) Program

Student in the MS program must turn in a thesis.

C1

L1   L2   L3

Student must submit three copies of thesis to thesis clerk.

C2

Student must pay binding fee.

C3

Student must file a thesis proposal in the graduate office.

C4

L4   L5   L6

Thesis must be printed on blue line paper.

C5

Thesis format must follow guidelines.

C6
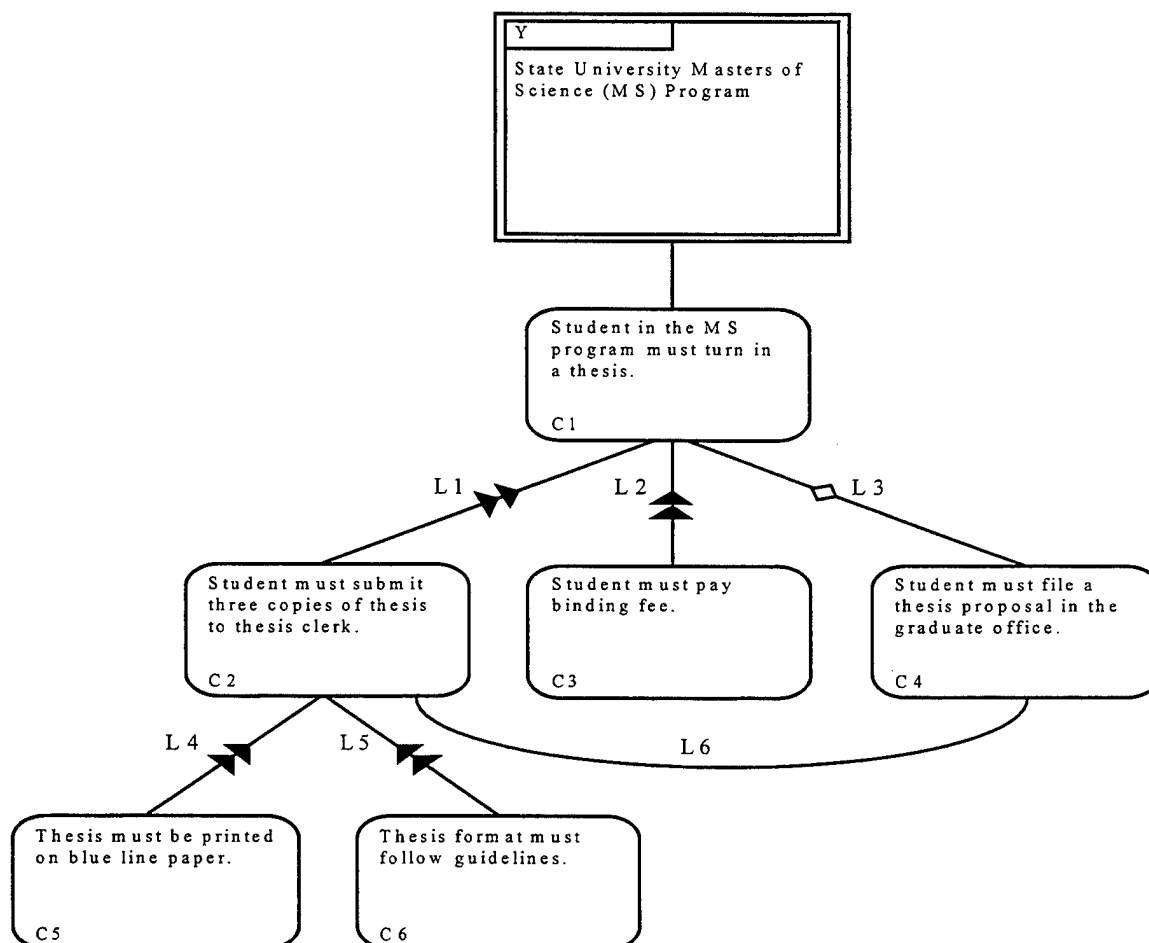
Figure A-56

Example Constraint Relationship Schematic

259
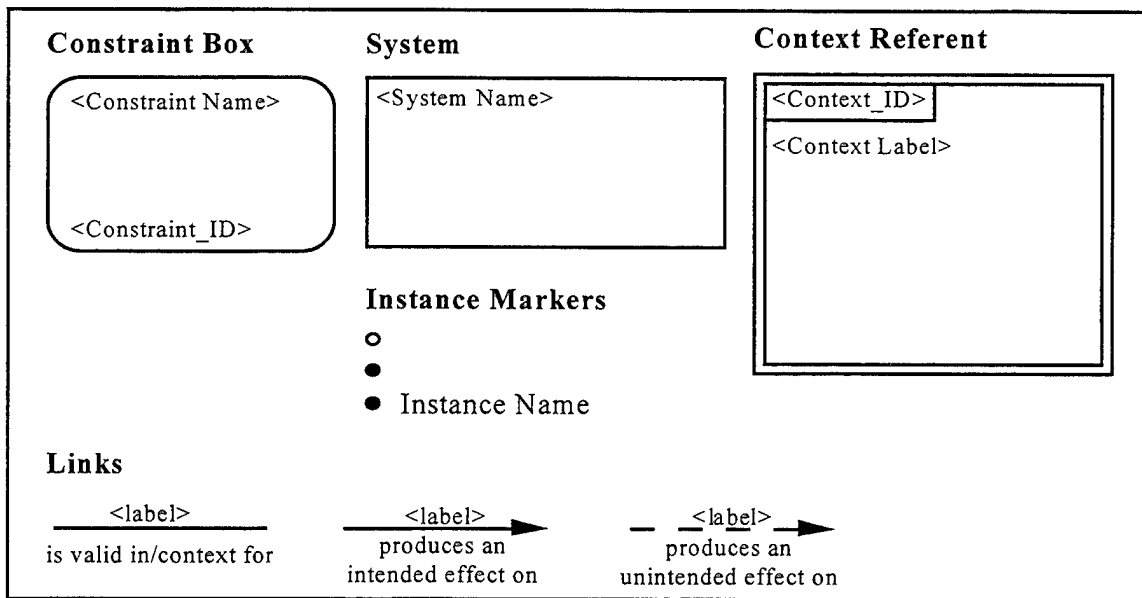
Figure A-57

Candidate Syntax for the Constraint Effects Schematic

These can be combined to illustrate that an economic order quantity purchasing constraint produces both intended and unintended affects (Figure A-58).
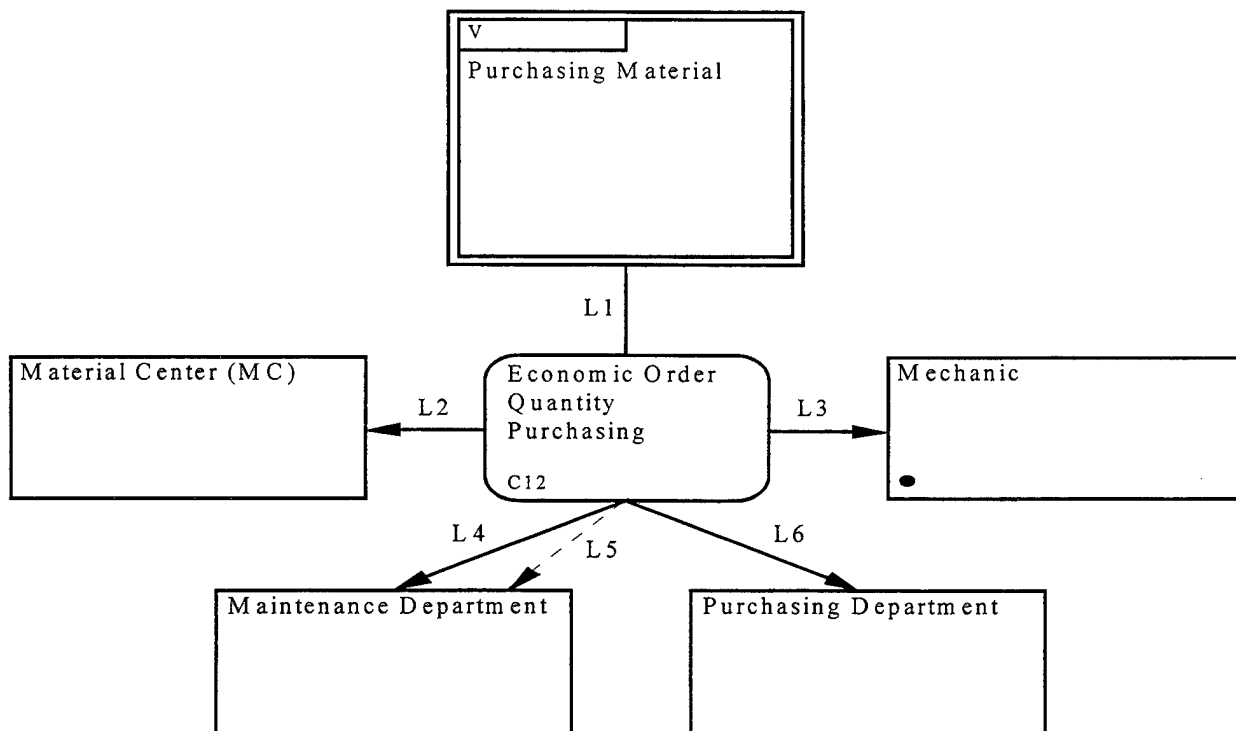
Figure A-58

Example Constraint Effects Schematic

Goal Schematic. The purpose of the Goal Schematic is to show the relationship between constraints and individual goals. Figure A-59 displays the candidate syntax for the Goal Schematic.
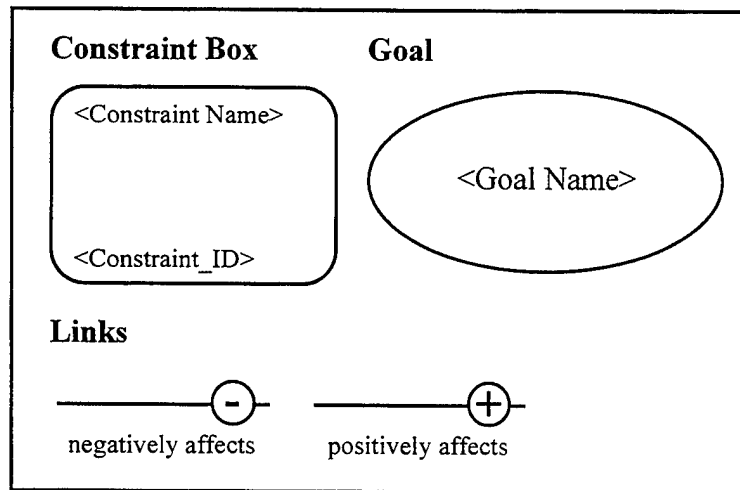
Figure A-59

Candidate Syntax for the Goal Schematic

Constraints that have a negative effect on the goal are displayed to the left of the goal; constraints that have a positive effect are displayed to the right (Figure A-60). The Goal Schematic may also be used to display both the relative impact of each constraint with respect to the goal and the "prioritization" of constraints with respect to the system's goals. The constraints are ordered from top to bottom according to "priority" (the highest priority being at the top).
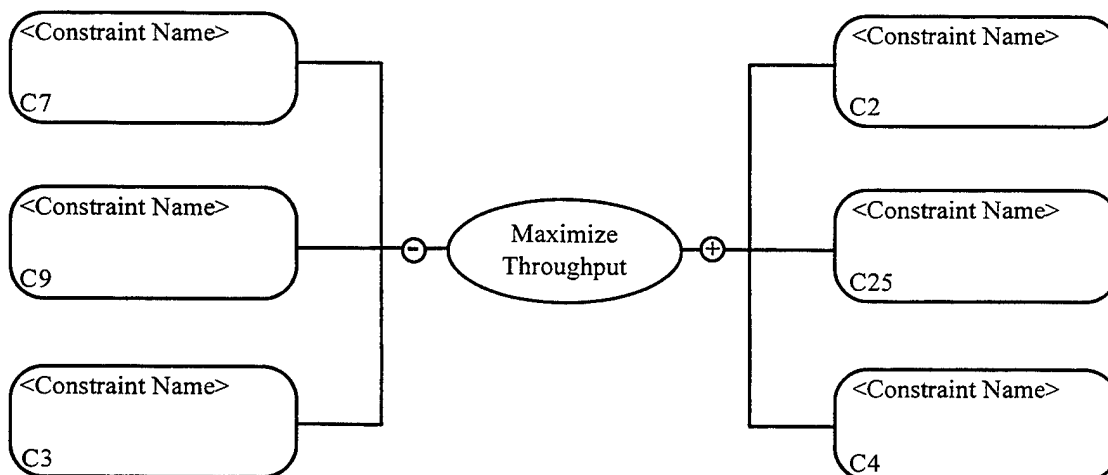
Figure A-60

Example Goal Schematic

Goal Relationship Schematic. The Goal Relationship Schematic is intended to show important relationships (subsumption, conflicts, etc.) between goals. A number of relationships between goals can be considered. Among these are *is valid in/context for*, *depends-on-existentially*, *implies*, *is-part-of*, and so forth. The concept of a performance measure may also be needed in the schematic (perhaps as a list of metric names in the goal object). No candidate designs for this schematic were proposed.

Symptom Schematic. The Symptom Schematic assists in tracing the underlying cause of symptoms to the lack of an enabling constraint or to the existence of a limiting constraint. The proposed syntax for this schematic is illustrated in Figure A-61.
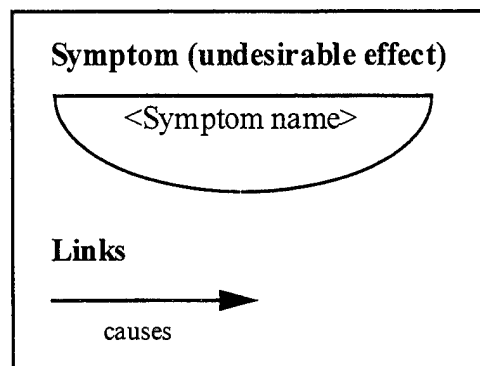
Figure A-61

Candidate Syntax for the Symptom Schematic

A symptom is defined as an indication or sign (e.g., change in normal function, sensation, or appearance) indicating a problem. A problem is defined as a source of trouble or annoyance. To one domain expert, another's problem is the cause of a new problem. To someone else, the same problem is viewed as a symptom of some other problem. Thus, negative effects are described differently (i.e., as the cause, problem, or effect) depending on the viewpoint taken. The actual root causes of negative effects (or positive effects) can be elusive. On the other hand, symptoms and causal relationships among symptoms are relatively easy for domain experts to recognize. Consequently, this schematic is called the *Symptom Schematic*. An example application of the Symptom Schematic is illustrated in Figure A-62.
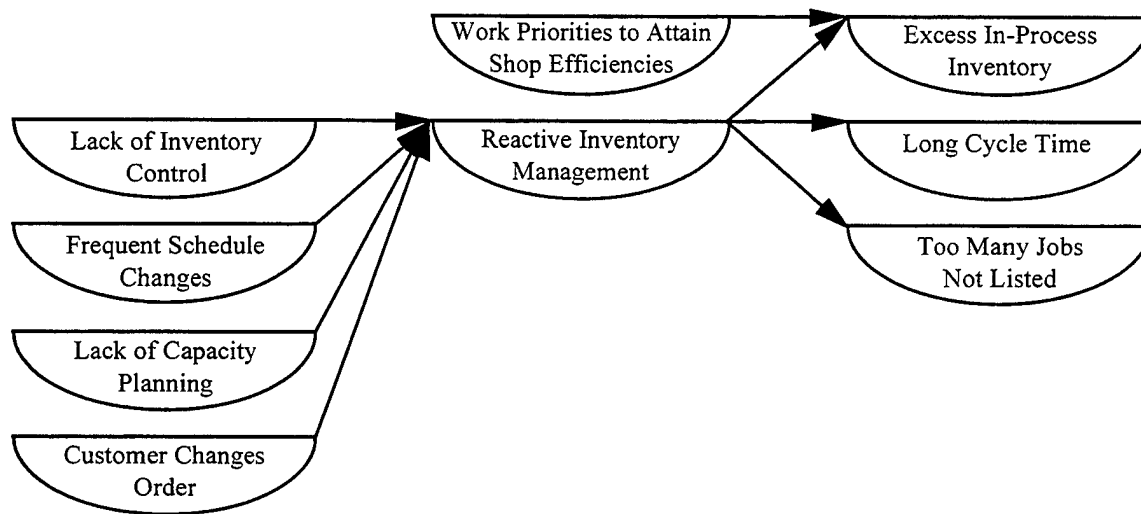
Figure A-62

Example Symptom Schematic

Summary. IDEF9 research has focused on developing a method to help users discover and map relevant constraints in an organizational system. Once these constraints have been cataloged, they can be examined systematically and tuned or replaced to improve the performance of the system. Significant progress has been made toward achieving these goals by developing a structured process for constraint discovery and the beginnings of a language supporting this process.

A more detailed description of the IDEF9 method can be found in the *IDEF Compendium* (Mayer et al., 1995c).

This page intentionally left blank.

APPENDIX B
A SAMPLE ONTOLOGY

The following is an ontology for the basic concepts of a bibliography developed by Thomas Gruber under funding from the Advanced Research Projects Agency (ARPA) for the Knowledge Sharing Effort (KSE).[47] It, rather than one of the ontologies developed under IICE, has been chosen as an illustration here because of its simple, nontechnical content. Also, unlike KBSI ontologies, which were developed primarily for internal use, Gruber's bibliography ontology is well documented because it was developed specifically to serve as an example ontology. The complete bibliography ontology, along with many others, can be viewed on the Stanford Knowledge Systems Laboratory's World Wide Web site (http://www-ksl.stanford.edu/knowledge-sharing/ontologies/). The ontology is specified in the language of the Knowledge Interchange Format (KIF), which is the basis of the IDEF5 elaboration language, supplemented by the Ontolingua *frame ontology*. Ontolingua was also developed in the KSE and provides a framework and software for sharing ontologies across different knowledge representation tools using (in general) different knowledge representation languages. The frame ontology embodies roughly the same conceptual categories as IDEF5, but differs somewhat in terminology; most notably, a *class* in the frame ontology is essentially an IDEF5 *kind*. The following introduction to Ontolingua definitions was also written by Gruber (though it has been edited slightly to enhance readability).

A Quick Introduction to Ontolingua Definitions

An ontology file is made up of definitions of classes, relations, functions, distinguished instances, and axioms that relate these terms. Classes are logically equivalent to unary relations, and functions are specializations of relations (which have one extra argument for the value). Classes, relations, and functions are also objects that can be arguments to certain second-order relations. These and other commitments are formalized and justified in the frame ontology.

A relation is defined with a form like the following:

```
(define-RELATION relation-name (?arg1 ?arg2)
  "documentation string"
  :def (and (domain-restriction ?arg1)
  (range-restriction ?arg2)
  (predicate-relating-args ?arg1 ?arg2)))
```

_____

[47] The bibiliography ontology is published on the World Wide Web, and can be found at the URL http://www-ksl.stanford.edu/knowledge-sharing/ontologies/html/bibliographic-data/bibliographic-data.lisp.html.

The sentence after the :def is a KIF sentence mentioning the arguments. Thus, one can define a relation by stating logical constraints over its arguments. These constraints are NECESSARY conditions that must hold if the relation holds over some arguments. It is also possible to state sufficient conditions, or any combination of necessary and sufficient conditions. Restrictions on the value of the first argument of a binary relation are effectively domain restrictions, and those on the second argument of a binary relation are range restrictions. There can be more than two arguments to a relation. Additional conditions on the arguments of a relation (expressed by means of complex logical expressions) that must hold among those arguments can also be added to the specification of a relation. Thus, a relation only holds between certain objects (i) when the argument restrictions are satisfied and (ii) all accompanying complex conditions are satisfied.

A class is defined with a very similar form; in fact, a class is just the case of a relation definition where there is exactly one argument, as in the following example.

```
(define-CLASS class-name (?instance)
  "documentation string"
  :def (and (superclass ?instance)
  (other-properties-of-instances ?instance)))
```

The :def sentence describes those conditions that must be met by instances of the class. Unary predicates applied to the instance variable specify superclasses of the class being defined. Other properties can be stated using KIF sentences mentioning the instance variable.

A function is defined like a relation, with a slight variation in syntax:

```
(define-FUNCTION function-name (?arg1 ?arg2) :-> ?value
  "documentation string"
  :def (and (argument-restriction ?arg1)
        (range-restriction ?value))
  :lambda-body (function1 (function2 ?arg1) (function3 ?arg2)))
```

The arguments to a function are constrained as in relations; the function is only defined on arguments that satisfy these constraints. A restriction on the range of the function is specified with a sentence constraining the value variable. A function of N arguments is equivalent to a relation of N+1 arguments with the additional constraint that there is at most one value for a given binding for the arguments.

When a side-effect-free functional expression is known for determining the value of a function from values of its arguments, it can be given as a KIF term expression following :lambda-body.

In any of the definitional forms, one may also write KIF sentences that help define the class, relation, or function but do not mention the argument variables. These are stand-alone axioms, labeled with the :axiom-def

keyword. Such axioms often use special second-order relations from the frame ontology that take functions, relations, and classes as arguments. For example,

```
(define-class C (?i)
    "The class C is a subclass of P and is divided into three disjoint subclasses X, Y, and Z."
    :def (P ?i)
    :axiom-def (exhaustive-subclass-partition C (setof X Y Z)))
```

This exhaustive-subclass-partition statement says that C is completely divided into three subclasses, which are mutually disjoint. This is much more succinct than the expanded KIF axiom:

```
(and (=> (C ?i) (or (X ?i) (Y ?i) (Z ?i)))
    (=> (X ?i) (C ?i))
    (=> (Y ?i) (C ?i))
    (=> (Z ?i) (C ?i))
    (=> (X ?i) (not (Y ?i)))
    (=> (X ?i) (not (Z ?i)))
    (=> (Y ?i) (not (X ?i)))
    (=> (Y ?i) (not (Z ?i)))
    (=> (Z ?i) (not (X ?i)))
    (=> (Z ?i) (not (Y ?i))))
```

The Bibliography Ontology[48]

This file defines an ontology for representing bibliographic data, such as those used in reference lists and document index databases. The objective of this ontology is to define the concepts and relationships that underlie a family of databases and tools in the domain of bibliographies. Such a conceptualization is intended to help with automatic translation among existing databases, to enable the development of reference-formatting styles that are independent of database or tool, and to support remote services such as bibliography database search and reference-list generation.

An ontology can be partitioned into theories. This file contains the GENERIC-BIBLIOGRAPHY, which establishes basic terminology. Child theories that include (specialize) the generic bibliography ontology will describe the constraints of specific bibliographic databases and tools.

---

[48] Version 1. Last modified 11 August 1992. Author: Thomas Gruber

267

Basic Ontological Commitments

A bibliography is made up of references. A REFERENCE describes the information needed to identify and retrieve a publication. A publication is associated with a DOCUMENT of some sort (e.g., a book or journal). In some cases there are several publications per document (e.g., papers in an edited collection). Thus, documents are distinguished from references. Documents are created by AUTHORs, which are PEOPLE or other agents (e.g., ORGANIZATIONs). They are published by PUBLISHERs or other organizations.

A very simple ontology of time is included. A TIMEPOINT is a specification of a single point in historical time. CALENDER-DATE is a timepoint at the resolution of days; that is, the day, month, and year are known. A CALENDER-YEAR is a timepoint at the resolution of years. The publication date of DOCUMENTs is some kind of timepoint; for many publications only the year is known. Events such as CONFERENCEs also occur on dates specified with timepoints.

All documents have titles. TITLEs are names (strings of characters), as are AGENT-NAMEs, CITY-ADDRESSes, and other data types that are used as identifiers and are not further destructured. The class called BIBLIO-NAME is a place holder for these atomic identifiers. The class BIBLIO-NL-TEXT is for strings of characters meant for human reading rather than as an identifier.

The most interesting ontological commitment is the distinction between the data fields in a REFERENCE and the facts about DOCUMENTs. Facts are stated as relationships over, and properties of, explicitly represented objects. For example, some facts are about publishers: publisher's name, city with which the publisher is associated, and year of (latest) publication. In a document, the DOC.PUBLISHER is an ORGANIZATION. In a REFERENCE, the REF.PUBLISHER is the PUBLISHER.NAME of the publisher, and the REF.ADDRESS is the PUBLISHER.ADDRESS of the document's publisher. The REF.YEAR of the reference is a number, which is the TIMEPOINT.YEAR of the DOC.PUBLICATION-DATE (a timepoint) of the document associated with a reference. Thus, in a reference — the entity we are trying to share — many of the facts have been mapped onto these atomic data types such as name strings, title strings, and numbers. In a document, some of the meaning of these data types can be stated as logical constraints. This is in the spirit of the CYC project, which aims to provide the background knowledge behind the narrow, brittle representations of expert systems and conventional databases.

This distinction between data in references and facts in other parts of the ontology supports the interoperability of tools that commit to this ontology, and the integration of this ontology and associated databases with other ontologies and databases. Part of the incompatibility between bibliography databases and tools is due to different *encodings* and *presentations* of the same information. For example, one database might encode a date as a string, another as a structured record of integer fields. Their ontological commitments might be the same — they might both support years and months for a magazine article reference — but their encodings mask their conceptual agreement. Similarly, different bibliography formatting styles might disagree on whether a publisher city is

required for a given reference type, but they agree that the city is a function of the publisher (not a name for it found in some reference field). Explicitly representing agents (authors), organizations (publishers), events (conferences), and time (publication dates) as objects in the conceptualization allows one to write knowledge-full constraints about how the data fields are related. Capturing these constraints is part of good database design because it reduces redundancy and anticipates integration with other databases. For example, associating author and publisher *names* (which are all that appear in references) with independently defined agents ensures that these agents will be named consistently in the references and facilitates the importation of data on authors and publishers from other databases.

This ontology is a set of definitions of classes, relations, and functions. It is represented in Ontolingua forms, which use KIF as the formal language and English documentation strings to describe meanings that haven't been formalized.

The definitions in the bibliography ontology are organized in a mildly bottom-up order, as follows:

1. Introduction
2. Generic Data Classes: biblio-name, biblio-NL-text, title, keyword
3. Agents and Events: persons, organizations, publishers, conferences
4. Dates and units of time
5. Documents
5.1 The document class and slots on it
5.2 Document types (subclasses): books, periodicals, theses, reports, etc.
6. References
6.1 The reference class and slots on it
6.2 Reference types (subclasses): book-ref, article-ref, thesis-ref, etc.
6.3 Constraints used frequently to define reference types
7. Acknowledgments

```
;;; ------------------------------------------------------------
;;; 1. Introduction
;;;
;;; First we need to establish the package in which this is to be ;;; read. Ontolingua-user is a good package to write
ontologies. ;;; It inherits the exported symbols from the KIF and Ontolingua ;;; packages, which gives the user use
of all the symbols defined ;;; in the KIF ontology (e.g., AND, OR, SETOF, =>) and Frame-
;;; ontology (e.g., VALUE-TYPE, HAS-ONE, SUBCLASS-PARTITION).

(in-package "ONTOLINGUA-USER")

;;; All ontologies need to be in one or more theories. Here we ;;; define a theory that includes just the frame
ontology.
;;; DEFINE-THEORY is analogous to DEFPACKAGE.
(define-theory GENERIC-BIBLIOGRAPHY (frame-ontology)
 "The generic-bibliography ontology defines the terms used for describing bibliographic references. This theory
defines the basic class for reference objects and the types (classes) for the data objects that appear in references,
such as authors and titles. Specific databases will use schemata that associate references with various combinations
of data objects, usually as fields in a record. This ontology is intended to provide the basic types from which a
specific database schema might be defined.")
```

;;;; DEFINE-THEORY only defines the theory, it doesn't make it the ;;; "current theory". The following form does just that. IN-
;;; THEORY is analogous to IN-PACKAGE. The rest of the ontolingua ;;; forms in this file are interpreted with respect to the
;;; GENERIC-BIBLIOGRAPHY theory.

(in-theory 'GENERIC-BIBLIOGRAPHY)

;;; --
;;; 2. Generic Data Classes: biblio-name, biblio-NL-text, title, ;;; keyword

;;; Ontologies will commonly have a top-level class such as the ;;; following. They are more for organization rather than
;;; meaning, but are useful for assimilating a small ontology into ;;; a more comprehensive ontology.

```
(define-class BIBLIO-THING (?x)
  "Biblio-thing is the root of the bibliographic ontology."
  :def (individual ?x)  ; from KIF ontology - means "not a set"
  :axiom-def (subclass-partition
BIBLIO-THING
(setof biblio-text
agent
organization
timepoint
document
reference)))
```

```
(define-class BIBLIO-TEXT (?string)
  "The most general class of undifferentiated text objects."
  :def (and (biblio-thing ?string)
(string ?string))
  :issues (("Why not make biblio-name and biblio-nl-text be
disjoint subclasses of biblio-text?"
"There may be valid names that are exactly the same strings
as NL texts, so it would be overconstraining to require
these two classes to be disjoint. The distinction between
names and NL texts is in their intended use, rather than
their forms. This happens because we don't represent what
the strings denote. If we did, then we could offer a
formal basis for distinguishing between these two classes
of text strings.")))
```

```
(define-class BIBLIO-NAME (?string)
  "A name of something in the bibliography ontology. Names are distinguished from strings in general because they
may be treated specially in some databases; for example, there may be uniqueness assumptions."
  :def (biblio-text ?string))
```

```
(define-class BIBLIO-NL-TEXT (?string)
  "A string of natural language text mentioned in some bibliographic reference. Texts are distinguished from strings
in general because they may be treated specially in some databases, or presented as free-flowing text to a human
reader. Biblio-texts are used for different purposes than biblio-names. Biblio texts are for things like notes and
abstracts; biblio-names are meant to identify some object or some property."
  :def (biblio-text ?string))
```

```
(define-class TITLE (?x)
  "A title is a string naming a publication, a document, or  something analogous.  Title strings are distinct from
strings naming agents (books can't talk)."
  :def (biblio-name ?x))

(define-class KEYWORD (?keyword)
  "A keyword is a name used as an index."
  :def (biblio-name ?keyword))

(define-class CITY-ADDRESS (?name)
  "A city-address is a string that identifies a city somewhere in the world.  We distinguish it from other names to
facilitate integrating it with ontologies that include representations for locations and alternative ways of identifying
places."
  :def (biblio-name ?name))


;;; ----------------------------------------------------------;;; 3. Agents and Events: persons, organizations, publishers,
;;; conferences.  People, organizations, and events involved in
;;; creating and publishing documents

(define-class AGENT (?x)
  "An agent is something or someone that can act on its own and produce changes in the world.  There is more to
agenthood than that, but for this ontology that is all that matters."
  :def (and (biblio-thing ?x)
(has-one ?x agent.name))
  :axiom-def (subclass-partition
AGENT
(setof person
organization))
  :issues ("could be merged with more meaningful definitions
of agenthood in larger ontologies."))

(define-function AGENT.NAME (?agent) :-> ?name
  "Function from an agent to the name by which it goes. One name per agent."
  :def (and (agent ?agent)
(biblio-name ?name)))

(define-class AGENT-NAME (?name)
  "A string that is the name of some agent."
  :axiom-def (exact-range AGENT.NAME AGENT-NAME))

(define-class PERSON (?x)
  "Human person"
  :def (agent ?x))

(define-function PERSON.NAME (?person) :-> ?name
  "The name that people use as authors of publications.  One name per person."
  :def (and (person ?person)
(biblio-name ?name)))

(define-class AUTHOR (?x)
  "An author is a person who writes things.  An author must have a name."
  :def (and (person ?x)
(has-one-of-type ?x author.name biblio-name)
(have-same-values ?x author.name person.name)))
```

```
(define-class ORGANIZATION (?x)
  "An organization is a corporate or similar institution, distinguished
from persons and other agents."
  :def (and (biblio-thing ?x)
(agent ?x)))

(define-function ORGANIZATION.NAME (?organization) :-> ?name
  "The name by which organizations go by.  One name per place."
  :def (and (organization ?organization)
(biblio-name ?name)))

(define-class PUBLISHER (?x)
  "A publisher is an organization that publishes."
  :def (and (organization ?x)
(has-one-of-type ?x PUBLISHER.NAME biblio-name)))

(define-function PUBLISHER.NAME (?publisher) :-> ?name
  "The name of a publisher; one per publisher."
  :def (and (publisher ?publisher)
(biblio-name ?name)))

(define-class PUBLISHER-NAME (?name)
  "A name of some publisher"
  :axiom-def (exact-range publisher.name PUBLISHER-NAME))

(define-relation PUBLISHER.ADDRESS (?publisher ?city)
  "The publisher.address is the name of a city with which a publisher is associated for document ordering purposes.
There may be several cities associated with a publisher.  If the city is well-known, then just its name is given;
otherwise its name and state and sometimes  country are given as the location."
  :def (and (publisher ?publisher)
(city-address ?city)))

(define-class UNIVERSITY (?x)
  "A university is an institute of higher learning that offers a graduate research program.  Of importance here is the
fact that universities sponsor the publication of dissertations."
  :def (organization ?x))

(define-class CONFERENCE (?x)
  "A conference is a big meeting where people wear badges, sit
through boring talks, and drink coffee in the halls."
  :def (and (biblio-thing ?x) ;should be an event of some kind
(has-one-of-type ?x conf.name biblio-name)
(has-one-of-type ?x conf.organization organization)
(has-one-of-type ?x conf.date calendar-date)
(value-type ?x conf.address city-address)
(can-have-one ?x conf.address)))


;;; ------------------------------------------------------------
;;;
;;; 4. Dates and units of time
;;; An incomplete ontology of time.  Should be merged with a more
;;; general-purpose version, such as those used for scheduling.

(define-class TIMEPOINT (?t)
```

"A timepoint is a specification of an absolute point in real, historical time."
:def (biblio-thing ?t))

(define-class CALENDER-YEAR (?t)
"a specification of a point in absolute calendar time, at the
resolution of one year."
:def (and (timepoint ?t)
(has-one ?t timepoint.year)))

(define-class CALENDER-DATE (?t)
"a specification of a point in absolute calendar time, at the resolution of one day."
:def (and (timepoint ?t)
(has-one ?t timepoint.day)
(has-one ?t timepoint.month)
(has-one ?t timepoint.year)))

(define-class UNIVERSAL-TIME-SPEC (?t)
"a specification of a point in real-world, historical, wall-clock time, independent of time zone and with one second
resolution."

:def (and (timepoint ?t)
(has-one ?t timepoint.seconds)
(has-one ?t timepoint.minutes)
(has-one ?t timepoint.day)
(has-one ?t timepoint.month)
(has-one ?t timepoint.year)))

(define-function TIMEPOINT.YEAR (?timepoint ?year) :-> ?year
"function from time points to integers representing the year component of the time specification. The integer
represents the number of years A.D., e.g., 1992."
:def (and (timepoint ?timepoint)
(year-number ?year)))

(define-class YEAR-NUMBER (?year)
"A year expressed as the number of years A.D."
:def (integer ?year))

(define-function TIMEPOINT.MONTH (?timepoint) :-> ?month
"function from time points to months, representing the month component of the time specification. Months are not
integers, but named objects."
:def (and (timepoint ?timepoint)
(month-name ?month)))

(define-class MONTH-NAME (?month)
"The months of the year, specified as an extensionally-defined (i.e., enumerated) set of objects, in English.
Instances of this class of months are not _symbols_, they are months that may be denoted by object constants."
:iff-def (member ?month
(setof january february march april may june july august
september october november december))
:issue (("Why not specify them as an ordered sequence?"
"The class only defines the set of months. Their order would
be given by an ordering predicate: a binary relation.")))

```
(define-function TIMEPOINT.DAY (?timepoint) :-> ?day-of-month
   "function from time points to integers representing the day component of the time specification."
   :def (and (timepoint ?timepoint)
   (integer ?day-of-month)
   (=< 0 ?day-of-month)
   (=< ?day-of-month 31)))


(define-function TIMEPOINT.MINUTES (?timepoint) :-> ?minutes
   "function from time points to integers representing the minutes component of the time specification."
   :def (and (timepoint ?timepoint)
   (integer ?minutes)
   (=< 0 ?minutes)
   (< ?minutes 60)))


(define-function TIMEPOINT.SECONDS (?timepoint) :-> ?seconds
   "function from time points to integers representing the seconds component of the time specification.  This is not
the internal representation of the universal time, (e.g., number seconds since some historical date).
Timepoint.seconds is the number of seconds past the minute, hour, day, etc. specified in the other components of the
timepoint."
   :def (and (timepoint ?timepoint)
   (integer ?seconds)
   (=< 0 ?seconds)
   (< ?seconds 60)))


;;; ------------------------------------------------------------
;;; 5. Documents
;;;
;;; 5.1 The document class and slots on it.

(define-class DOCUMENT (?x)
   "A document is something created by author(s) that may be viewed, listened to, etc., by some audience.  A
document persists in material form (e.g., a concert or dramatic performance is not a document).  Documents
typically reside in libraries."
   :def (and (biblio-thing ?x)
   (has-one ?x doc.title))
   :axiom-def (subclass-partition
DOCUMENT
(setof book
periodical-publication
proceedings
thesis
report
miscellaneous-publication)))


(define-function DOC.TITLE (?doc) :-> ?title
   "The title of a document.  Not necessarily the title of a work published in the document."
   :def (and (document ?doc)
   (title ?title)))

(define-relation DOC.AUTHOR (?doc ?agent) ?name
   "The creator(s) of a document.  Not necessarily the author of a work published in the document, but often so.  The
author is a real agent, not a name of an agent."
   :def (and (document ?doc)
   (agent ?agent)))
```

```
(define-function DOC.PUBLICATION-DATE (?doc) :-> ?year
  "The publication date of a document. If the document isn't formally published, e.g., a painting, then it is the year
of creation. The date is a timepoint for which at least the year is known. In some cases, the month and day are also
known."
 :def (and (document ?doc)
 (calendar-year ?year))
 :issues ("Calendar-date is a subclass-of calendar-year,
because it requires the same constraint (that a year be known)
plus a few more (that a month and day be known).  This means
that if we require publication dates to be calendar-years,
then we also allow more precise timepoint specifications
as well.  Thus, documents for which a month field is needed
can get the month from the publication date."))

(define-function DOC.PUBLISHER (?doc) :-> ?publisher
  "The publisher of a document, if there is one.  This is the  publisher, not its name."
 :def (and (document ?doc)
 (publisher ?publisher)))

(define-relation DOC.EDITOR (?doc ?editor)
  "Named primary editors of a document."
 :def (and
 (or (book ?doc)
 (proceedings ?doc))
 (person ?editor)))

(define-relation DOC.SERIES-EDITOR (?doc ?editor)
  "Series editors of a document."
 :def (and
 (or (book ?doc)
 (proceedings ?doc))
 (person ?editor)))

(define-function DOC.SERIES-TITLE (?doc) :-> ?title
  "Series title of a document."
 :def (and
 (or (book ?doc)
 (proceedings ?doc))
 (title ?title)))

(define-relation DOC.TRANSLATOR (?doc ?translator)
  "Named primary editors of a document."
 :def (and
 (or (book ?doc)
 (proceedings ?doc))
 (person ?translator)))

(define-function DOC.EDITION (?doc) :-> ?nth
  "Refers to the nth edition of a document."
 :def (and
 (or (book ?doc)
 (proceedings ?doc)
 (cartographic-map ?doc))
```

(natural-number ?nth)))

(define-function DOC.NUMBER-OF-PAGES (?doc) :-> ?n
"Number of pages contained a document. Not the page numbers of an article"
:def (and (document ?doc)
(natural-number ?n)))

;;; --
;;;; 5.2 Document types (subclasses): books, periodicals, theses,
;;;; proceedings, reports, maps, manuals, etc.
;;;;
;;;

(define-class BOOK (?x)
"pages in a bound cover. You can't judge it by its cover."
:def (and (document ?x)
(has-some ?x doc.author)
(has-one ?x doc.title)
(has-one ?x doc.publication-date)
(has-one ?x doc.publisher)
))

(define-class EDITED-BOOK (?x)
"An edited book is a book whose authors are known as editors."
:iff-def (and (book ?x)
(has-some ?x doc.editor)
(have-same-values ?x doc.editor doc.author)))

(define-class PERIODICAL-PUBLICATION (?x)
"A periodical-publication is published regularly, such as once every week. Strictly speaking, the noun 'periodical'
is used by librarians to refer to things published at intervals of greater than a day. We use the phase periodical-
publication to include newspapers and other daily publications, since they share many bibliographic features."
:def (document ?x)
:axiom-def (subclass-partition
PERIODICAL-PUBLICATION
(setof journal
magazine
newspaper)))

(define-class JOURNAL (?x)
"A journal is an archival periodical publication. Note that a journal is not the same as a journal article or a
reference to an article. A journal is a document; a particular issue of a journal may contain several articles."
:def (periodical-publication ?x))

(define-class MAGAZINE (?x)
"A magazine is a periodical publication that is considered to be of more general interest than a journal."
:def (periodical-publication ?x))

(define-class NEWSPAPER (?x)
"A newspaper is a periodical publication that may be published as frequently as once a day."
:def (periodical-publication ?x))

(define-class PROCEEDINGS (?x)

"The published proceedings of a conference, workshop, or similar meeting. If the proceedings appear as an edited book, the document is an edited book with a title other than 'proceedings of...' Proceedings may have editors, however."
 :def (and (document ?x)
 (has-one-of-type ?x doc.conference conference)
 (have-same-values ?x doc.title
 (compose* conf.name doc.conference))
 (have-same-values ?x doc.publication-date
 (compose* conf.date doc.conference)))
 :issues ("We are assuming that the title of the proceedings is
 the same as the name of the conferences. This may be bogus."))

(define-class THESIS (?x)
 "An official report on a bout of graduate work for which one receives a degree, published by the university. Never mind that some fields make a big deal about the difference between dissertations and theses. From the bibliographic perspective, they are both of the same family."
 :def (and (document ?x)
 (has-one ?x doc.author)
 (has-one ?x doc.title)
 (has-one ?x doc.publication-date)
 (has-one-of-type ?x diss.university university))
 :axiom-def (exhaustive-subclass-partition
 THESIS
 (setof masters-thesis
 doctoral-thesis))
 :issues (("What about theses published as technical reports?"
 "If it is cited as a TR, it is a TR."
 "One can imagine adding a cross-reference field to
 references that points to associated documents or
 other references.")))

(define-class MASTERS-THESIS (?x)
 "M.S. thesis document."
 :def (thesis ?x))

(define-class DOCTORAL-THESIS (?x)
 "Ph.D. thesis document"
 :def (thesis ?x))

(define-class TECHNICAL-REPORT (?x)
 "A technical report is a paper published by some research organization."
 :def (and (document ?x)
 (has-some ?x doc.author)
 (has-one ?x doc.title)
 (has-one ?x doc.publication-date)
 (has-one-of-type ?x doc.institution organization)))

(define-class MISCELLANEOUS-PUBLICATION (?x)
 "A miscellaneous category of documents that are infrequently found in bibliographic references."
 :def (and (document ?x)
 (has-one ?ref doc.title))
 :axiom-def (subclass-partition
 MISCELLANEOUS-PUBLICATION
 (setof manual

```
computer-program
artwork
cartographic-map
multimedia-document)))

(define-class TECHNICAL-MANUAL (?x)
  "The doc.author is the technical writer."
  :def (miscellaneous-publication ?x))

(define-class COMPUTER-PROGRAM (?x)
  "The doc.author is the programmer."
  :def (miscellaneous-publication ?x))

(define-class ARTWORK (?x)
  "The doc.author is the artist."
  :def (and (has-one ?ref doc.author)
(miscellaneous-publication ?x)))

(define-class CARTOGRAPHIC-MAP (?x)
  "The doc.author is the cartographer."
  :def (miscellaneous-publication ?x))

(define-class MULTIMEDIA-DOCUMENT (?x)
  "A multimedia document is some identifiable communique in a modality such as audio, video, animation, etc. The
document may exist in digital form in a library that is accessed by electronic means."
  :def (miscellaneous-publication ?x))


;;; -----------------------------------------------------------------
;;; 6. References
;;; 6.1 The reference class and slots on it
;;;

(define-class REFERENCE (?ref)
  "A bibliographic reference is a description of some publication that uniquely identifies it, providing the
information needed to retrieve the associated document.  A reference is distinguished from a citation, which occurs
in the body of a document and points to a reference.  Note that references are distinguished from documents as well.
The information associated with a reference is contained in data fields, which are binary relations (often unary
functions).  A reference should at least contain information about the author, title, and year.  (Since there are
exceptions, that constraint is associated with a specialization of this class.) ."
  :def (biblio-thing ?ref)
  :axiom-def (exhaustive-subclass-partition
REFERENCE
(setof publication-reference
non-publication-reference)))
;;; Fields of references are represented as binary relations
;;; (slots).  If they are single-valued, they are defined as unary
;;; functions.  (Recall that in KIF, unary functions are a
;;; subclass of binary relations.)
(define-function REF.DOCUMENT (?ref) :-> ?document
  "Function from references to associated documents.  Is only defined on publication-references, since by definition
they are the references associated with documents."
  :def (and (publication-reference ?ref)
(document ?document)))
```

278

```
(define-relation REF.AUTHOR (?ref ?author-name)
  "Relation between a reference and the name(s) of the
creator(s) of the publication."
  :def (and (reference ?ref)
  (agent-name ?author-name)))

(define-function REF.TITLE (?ref) :-> ?title
  "Most general relation between a reference and the PRIMARY title of a publication. The primary title goes by
various names depending on the reference type."
  :def (and (reference ?ref)
  (title ?title)))

(define-function REF.YEAR (?ref) :-> ?year
  "The year field is a function from a reference to the year in which the publication was published."
  :def (and (reference ?ref)
  (year-number ?year)))

(define-function REF.PERIODICAL (?ref) :-> ?journal-title
  "Most general relation between a reference and a journal."
  :def (and (reference ?ref)
  (title ?journal-title)))

(define-function REF.VOLUME (?ref) :-> ?number
  "in a reference, the volume number of a journal or magazine in which an article occurs."
  :def (and
  ;; domain is a subclass of all references
  (or
  (book-reference ?ref)
  (book-section-reference ?ref)
  (article-reference ?ref))
  ;; range is natural numbers
  (natural-number ?number)))

(define-function REF.ISSUE (?ref) :-> ?issue-number
  "In a reference, the issue number of a journal or magazine in which an article occurs."
  :def (and (article-reference ?ref)
  (natural-number ?issue-number)))

(define-function REF.REPORT-NUMBER (?ref) :-> ?identifier
  "An alphanumeric identifier that identifies a technical report within a series sponsored by the publishing
institution.  For example, STAN-CS-68-118 is the 118th report number of a report written at Stanford in the
computer science department in 1968."
  :def (and (technical-report-reference ?ref)
  (biblio-name ?identifier)))

(define-function REF.PAGES (?ref) :-> ?page-range
  "In a reference, the pages of an article or analogous subdocument in which a publication resides.  Specified as a
sequence of two integers."
  :def (and
  ;; domain is the union of these ref types:
  (or (book-section-reference ?ref)
  (article-reference ?ref)
  (proceedings-paper-reference ?ref))
  ;; range is a list of (starting ending) pages
```

```
(list ?page-range)
(integer (first ?page-range))
(integer (second ?page-range))))
```

```
(define-function REF.MONTH (?ref) :-> ?month
 "In a reference, the month in which a publication is published. Useful for magazines, conference proceedings, and
technical reports."
 :def (and (or (article-reference ?ref)
(technical-report ?ref))
(month-name ?month)))
```

```
(define-function REF.DAY (?ref) :-> ?month
 "In a reference, the day of the month in which a publication is published. Useful for conference proceedings,
personal communications."
 :def (and (or (article-reference ?ref)
(personal-communication-reference ?ref))
(day ?month)))
```

```
(define-relation REF.NOTES (?ref ?note-string)
 "In a reference, the notes field contains a set of strings that is used to describe all sorts of things."
 :def (and (reference ?ref)
(biblio-NL-text ?note-string)))
```

```
(define-function REF.ABSTRACT (?ref) :-> ?abstract-string
 "In a reference, the abstract field contains a string of natural language text that is used to describe all sorts of
things."
 :def (and (reference ?ref)
(biblio-NL-text ?abstract-string)))
```

```
(define-relation REF.KEYWORDS (?ref ?keyword)
 "Keywords associated with a reference."
 :def (and (reference ?ref)
(keyword ?keyword)))
```

```
(define-relation REF.LABELS (?ref ?label)
 "Labels associated with a reference."
 :def (and (reference ?ref)
(biblio-name ?label)))
```

```
(define-relation REF.EDITOR (?ref ?editor)
 "A reference's editor is the name of the document's editor."
 :axiom-def (= REF.EDITOR
(compose* author.name doc.editor ref.document)))
```

```
(define-relation REF.SERIES-EDITOR (?ref ?editor)
 "A reference's series editor is the name of a series editor of
the document."
 :axiom-def (= REF.SERIES-EDITOR
(compose* author.name doc.series-editor ref.document)))
```

```
(define-relation REF.TRANSLATOR (?ref ?translator)
 "A reference's translator is the name of the document's translator."
 :axiom-def (= REF.TRANSLATOR
(compose* author.name doc.translator ref.document)))
```

```
(define-relation REF.SECONDARY-AUTHOR (?ref ?editor)
  "In a reference, the secondary author field usually names an editor of some sort who was involved in the
production of the work but who was not a original author."
  :def (and (reference ?ref)
  (agent-name ?author-name)))

(define-relation REF.SECONDARY-TITLE (?ref ?title)
  "In a reference, the secondary title usually names the book or  serial in which the publication is published."
  :def (and (reference ?ref)
  (title ?title)))

(define-relation REF.ADDRESS (?ref ?address)
  "The place (e.g., city) where a document is published.  Means different things depending on the reference type."
  :def (and (reference ?ref)
  (city-address ?address)))

(define-function REF.PUBLISHER-NAME (?ref) :-> ?publisher-name
  "The publisher field of a reference points to the publisher of  the associated document."
  :def (and (reference ?ref)
  (publisher-name ?publisher-name)))

(define-function REF.NUMBER-OF-VOLUMES (?ref) :-> ?number
  "In a reference, the number of volumes in the associated document."
  :def (and (reference ?ref)
  (natural-number ?number)))

(define-function REF.EDITION (?ref) :-> ?nth
  "Refers to the nth edition of a document."
  :def (and (reference ?ref)
  (natural-number ?nth))
  ;; ref's edition is the document's edition
  :lambda-body (doc.edition (ref.document ?ref)))

(define-function REF.TYPE-OF-WORK (?ref) :-> ?name
  "An identifier of some specialization within the reference type. For example, technical reports are labeled with
types-of-work such as 'technical report' and 'memo'.  Dissertations are specialized by the level of the associated
degree."
  :def (and (or (thesis-reference ?ref)
  (technical-report-reference ?ref)
  (miscellaneous-reference ?ref))
  (biblio-name ?name)))


;;; -----------------------------------------------------------
;;; Reference types

(define-class PUBLICATION-REFERENCE (?ref)
  "A reference associated with some kind of published document, where publication and documenthood are
interpreted broadly."
  :def (and (has-one-of-type ?ref ref.document document)
  (has-one ?ref ref.title))
  :axiom-def (subclass-partition
  PUBLICATION-REFERENCE
  (setof book-reference
```

book-section-reference
article-reference
proceedings-paper-reference
thesis-reference
technical-report-reference
misc-publication-reference)))

(define-class NON-PUBLICATION-REFERENCE (?ref)
 "A reference to something that just isn't a document."
 :def (and (publication-reference ?ref)
(cannot-have ?ref ref.document))
 :axiom-def (subclass-partition
NON-PUBLICATION-REFERENCE
(setof personal-communication-reference
generic-unpublished-reference)))

(define-class BOOK-REFERENCE (?ref)
 "A book reference. Book references usually include complete publisher information, and may have a series editor
and title, edition, and translator. A reference to a book gets many of its publication data from the book qua
document."
 :def (and (publication-reference ?ref)
        (has-one-of-type ?ref ref.document book)
        (inherits-author-from-document ?ref)
        (inherits-title-from-document ?ref)
        (book-publication-data-constraint ?ref) ;see-below
        (have-same-values ?ref ref.secondary-author ref.series-
         editor)
        (have-same-values ?ref ref.secondary-title doc.series-
         title)))

(define-class EDITED-BOOK-REFERENCE (?ref)
 "Like a book-reference, except the document is an edited-book and the author and editor are the same."
 :iff-def (and (book-reference ?ref)
(has-one-of-type ?ref ref.document edited-book)
(has-some ?ref ref.editor)
(have-same-values ?ref ref.author ref.editor)))

(define-class BOOK-SECTION-REFERENCE (?ref)
 "A section of a book, like a chapter or a paper in an edited collection."
 :def (and (publication-reference ?ref)
(has-one-of-type ?ref ref.document edited-book)
(has-some ?ref ref.author)
(has-some ?ref ref.editor)
(has-one ?ref ref.booktitle)
(have-same-values ?ref ref.booktitle
(compose* doc.title ref.document))
(book-publication-data-constraint ?ref)
(have-same-values ?ref ref.secondary-author ref.editor)
(have-same-values ?ref ref.tertiary-author ref.series-editor)))

(define-class ARTICLE-REFERENCE (?ref)
 "An article is a piece published in a journal, magazine, or newspaper."
 :def (and (publication-reference ?ref)
(value-type ref.document periodical-publication)

282

```
(has-some ?ref ref.author)
(has-one ?ref ref.title)
(has-one ?ref ref.year)
(has-one ?ref ref.periodical)
(have-same-values ?ref ref.periodical
(compose* doc.title ref.document))
(have-same-values ?ref ref.secondary-title
ref.periodical)))

(define-class JOURNAL-ARTICLE-REFERENCE (?ref)
  "A reference to article in a journal must give information sufficient to find the issue containing the article."
  :def (and (article-reference ?ref)
(value-type ref.document journal)
(cannot-have ?ref ref.month)  ; unlike other articles
))

(define-class MAGAZINE-ARTICLE-REFERENCE (?ref)
  "A reference to an article in a magazine is essentially the same as a journal article reference.  Some formatting
styles need the distinction. Magazine article references sometimes include the month instead of the volume/issue
numbers."
  :def (and (article-reference ?ref)
(value-type ref.document magazine)
(has-one ?ref ref.magazine-name)
(have-same-values ?ref ref.magazine-name
ref.periodical)))

(define-class NEWSPAPER-ARTICLE-REFERENCE (?ref)
  "A newspaper article reference is like a magazine article reference"
  :def (and (article-reference ?ref)
(value-type ref.document newspaper)
(has-one ?ref ref.newspaper-name)
(have-same-values ?ref ref.magazine-name ref.periodical)
(has-one ?ref ref.month)
(has-one ?ref ref.day)
(value-type ?ref ref.address city-address)))

(define-class PROCEEDINGS-PAPER-REFERENCE (?ref)
  "An article appearing in the published proceedings of some conference or workshop."
  :def (and (publication-reference ?ref)
(value-type ref.document proceedings)
(has-some ?ref ref.author)
(has-one ?ref ref.title)
(inherits-year-from-document ?ref)
(has-one ?ref ref.booktitle)
(have-same-values ?ref ref.booktitle
(compose* doc.title ref.document))
(have-same-values ?ref ref.secondary-title ref.booktitle)
(have-same-values ?ref ref.secondary-author ref.editor)
(have-same-values ?ref ref.organization
(compose* conf.organization
 doc.conference ref.document))
(have-same-values ?ref ref.address
(compose* conf.address
 doc.conference ref.document))
```

283

```
(have-same-values ?ref ref.month
(compose* timepoint.month
 doc.publication-date ref.document))
(have-same-values ?ref ref.day
(compose* timepoint.day
 doc.publication-date ref.document))))

(define-class THESIS-REFERENCE (?ref)
 "A reference to a master's or doctoral thesis."
 :def (and (publication-reference ?Ref)
 (value-type ref.document thesis)
 (inherits-author-from-document ?ref)
 (inherits-title-from-document ?ref)
 (inherits-year-from-document ?ref)
 (has-one ?ref ref.publisher)
 (has-same-values ?ref ref.publisher
 (compose* organization.name diss.university)))
 :axiom-def (subclass-partition
 THESIS-REFERENCE
 (setof doctoral-thesis-reference
 masters-thesis-reference)))

(define-class DOCTORAL-THESIS-REFERENCE (?ref)
 :def (and (thesis-reference ?ref)
 (value-type ref.document doctoral-thesis)
 (has-value ?ref ref.type-of-work doctoral-thesis)))

(define-class MASTERS-THESIS-REFERENCE (?ref)
 :def (and (masters-thesis ?ref)
 (value-type ref.document masters-thesis)
 (has-value ?ref ref.type-of-work masters-thesis)))

(define-class TECHNICAL-REPORT-REFERENCE (?ref)
 :def (and (publication-reference ?ref)
 (inherits-author-from-document ?ref)
 (inherits-title-from-document ?ref)
 (inherits-year-from-document ?ref)
 (has-one ?ref ref.publisher)
 (has-same-values ?ref ref.publisher
 (compose* organization.name doc.institution))))

(define-class MISC-PUBLICATION-REFERENCE (?ref)
 :def (publication-reference ?ref)
 :axiom-def (subclass-partition
 MISC-PUBLICATION-REFERENCE
 (setof technical-manual-reference
 computer-program-reference
 cartographic-map-reference
 artwork-reference
 multimedia-document-reference)))

(define-class TECHNICAL-MANUAL-REFERENCE (?ref)
 "A reference to a manual that may accompany a product but is otherwise unpublished."
 :def (and (misc-publication-reference ?ref)
```

```
(inherits-author-from-document ?ref)
(inherits-title-from-document ?ref)
(inherits-year-from-document ?ref)))

(define-class COMPUTER-PROGRAM-REFERENCE (?ref)
  "A reference to a computer program.  The ref.title is the name of the program.  The author is the programmer."
  :def (and (misc-publication-reference ?ref)
  (inherits-author-from-document ?ref)
  (inherits-title-from-document ?ref)))

(define-class CARTOGRAPHIC-MAP-REFERENCE (?ref)
  "A reference to a map created by a cartographer."
  :def (and (misc-publication-reference ?ref)
  (inherits-author-from-document ?ref)
  (inherits-title-from-document ?ref)
  (inherits-year-from-document ?ref)))

(define-class ARTWORK-REFERENCE (?ref)
  "A reference to a work of art that does not fit the other categories of documents.  The author is the artist."
  :def (and (misc-publication-reference ?ref)
  (inherits-author-from-document ?ref)
  (inherits-title-from-document ?ref)
  (inherits-year-from-document ?ref)))

(define-class MULTIMEDIA-DOCUMENT-REFERENCE (?ref)
  "A bibliographic reference to a multimedia document. Who knows what conventions the future holds for these
things."
  :def (and (misc-publication-reference ?ref)
  (inherits-author-from-document ?ref)
  (inherits-title-from-document ?ref)
  (inherits-year-from-document ?ref)))

(define-class PERSONAL-COMMUNICATION-REFERENCE (?ref)
  "A reference to a personal communication between the author of the paper in which the bibliography appears and
some other person.  The ref.author of the reference is the person with whom the conversation was held."
  :def (and (non-publication-reference ?ref)
  (has-one ?ref ref.author)
  (has-one ?ref ref.year)
  (has-one ?ref ref.month)
  (has-one ?ref ref.day)))

;;; ------------------------------------------------------------
;;;
;;;
;;;; 6.3 Constraints used frequently to define reference types.
;;;
;;;
;;;; Technically, the following are classes.  In intent, they are
;;;; constraints over slot values of classes (thus, they are
;;;; specializations of classes).  We use define-relation instead
;;;; of define-class for such constraints, but
;;;; there is no logical distinction between the unary-relation qua
;;;; class and qua constraint over class membership.

(define-relation INHERITS-AUTHOR-FROM-DOCUMENT (?ref)
```

"When a reference is a one-to-one account of a document, then the author in the reference (ref.author) is the name of the author of the document. This relation captures this relationship."
:iff-def (and (publication-reference ?ref)
(have-same-values ?ref ref.author
(compose*
author.name doc.author ref.document))))

(define-relation INHERITS-PUBLISHER-FROM-DOCUMENT (?ref)
"When a reference is a one-to-one account of a document, then the publisher in the reference (ref.publisher) is the name of the publisher of the document. This relation captures this relationship. Inherits the publisher's address as well."
:iff-def (and (publication-reference ?ref)
(have-same-values ?ref ref.publisher-name
(compose* publisher.name doc.publisher
ref.document))
(have-same-values ?ref ref.address
(compose* publisher.address
doc.publisher ref.document))))

(define-relation INHERITS-YEAR-FROM-DOCUMENT (?ref)
"When a reference is a one-to-one account of a document, then the year in the reference (ref.year) is the year of publication of the document. This relation captures this relationship."
:iff-def (and (publication-reference ?ref)
(have-same-values ?ref ref.year
(compose* timepoint.year
doc.publication-date))))
(define-relation INHERITS-TITLE-FROM-DOCUMENT (?ref)
:iff-def (and (publication-reference ?ref)
(have-same-values ?ref ref.title
(compose* doc.title ref.document))))

(define-relation BOOK-PUBLICATION-DATA-CONSTRAINT (?ref)
"In references associated with books, the reference fields for publication data such as publisher, place, and edition are all taken from the data on the book-document itself. This unary relation captures these constraints in one place, so that each of the book reference types can just inherit them."
:iff-def (and
;; the constraint applies to references
(publication-reference ?ref)
;; whose associated document is a book of some sort
(has-one-of-type ?ref ref.document book)
;; must list a publisher.
(has-one ?ref ref.publisher)
;; ref's publisher is the book publisher's name
;; and the address is the publisher's address
(inherits-publisher-from-document ?ref)
;; books must have a year, and the are the year of publication
;; of the book document.
(has-one ?ref ref.year)
(inherits-year-from-document ?ref)))

APPENDIX C
ACCOUNTS OF INTEGRATION

Integration in General

Zoetekouw (1992) defines integration as the elimination of the artificial barriers in the system. Integration includes — but goes beyond — improving coordination among components by increasing the interconnectivity in the system.

Integrated Information System and Information-Integrated System

In IICE: Program Foundations and Philosophy (Painter, 1991), an *information system* is defined as "a system comprised of both automated and non-automated components responsible for capturing, managing, and controlling information resources" and an *integrated information system* as a system that "allows for the sharing of common data and responds gracefully to change (physical and logical)." Physical change requires the system to be able to handle the replacement of a physical component with minimum disruption to the logical components. Logical change necessitates the capability to change the information requirements and, possibly, enlarge or narrow the scope of the information system.

An enterprise whose information system is integrated may not necessarily be integrated. Even though the information within the information system may be well-coordinated, the enterprise as a whole may not be well-coordinated with its information system. An **information-integrated** enterprise, on the other hand, is an enterprise that achieves integration through information integration. Such a system uses information about itself to support the flow of work. An important advantage of an information-integrated system is that it provides a natural framework for concurrent engineering and Total Quality Management.

Enterprise Integration

Microelectronics and Computer Technology Corporation (MCC) defines Enterprise Integration (EI) as "the application of integrating solutions, both cultural and technical, to improve an organization's business or mission" (MCC, 1991, p. 3). MCC believes that computer technology plays a major role in EI. Therefore, its efforts are concentrated on providing computer systems-integrated enterprises. MCC's integration emphasizes the coupling of models of the enterprise with the enterprise itself. It suggests that efforts in EI should include standards development, information capture in a more integrated fashion, ways to apply models directly to ensure "model-to-execution" integrity, and, finally, a measure of integration progress.

In summary, MCC's perspective is that EI is concerned essentially with computing technology and the ability of coupling enterprise models to the enterprise. An "integrating system" should be capable of providing both

feedback from the enterprise to the models and the transformation information necessary for rendering the models and the enterprise consistent. A high-level view of an integrated system is composed of three main elements: the enterprise models, the enterprise, and the EI system.

MCC 1991 also presents some principles of EI. The most important principles are summarized below:

- **Heterogeneity:** No EI solution can impose homogeneity on the systems (agents) to be integrated. Hence, the system should support a federated service infrastructure and a federated structure for models (i.e., a structure that integrates services and models by linking them in ways that do not require them to be altered in any intrinsic fashion). A federated structure, as it were, builds bridges between existing services and models rather than renovating or re-creating them in a different form. One way to achieve such federation is through the concepts of services and service protocols. Using a federated approach implies that the integrated systems are loosely coupled. The object-oriented paradigm is well-suited to this concept of loosely coupled systems.

- **Basic capabilities:** The set of basic capabilities that an EI computing infrastructure should support needs to be defined.

- **Reactive System:** EI solutions need to be reactive. In a reactive system, any entity may connect to any other entity and react to changes in the system without being aware of the "big picture." Reactive systems, compared to prescriptive systems, are more flexible. The "big picture" does not need to be updated when changes occur in the system. Only the agents involved or affected by the changes need to "react" to these changes.

- **Meta-Information:** Because integration involves the coordination of information and services in all parts of an enterprise, there must be models of the information system itself within the system.

- **Flexibility:** All EI solutions should be scaleable, extensible, adaptable, scopable, and secure.

Finally, MCC 1991 underlines that the notion of information flow is "perhaps the most important technical issue and is probably the key to any EI solution."

Fox on Enterprise Integration

The Toronto Virtual Enterprise (TOVE) is a virtual company with the purpose of providing an EI testbed. The project is concerned mainly with enterprise modeling as a means toward EI. The project focuses on

288

communication of information, coordination of decisions and persons, and optimization of the decision process. TOVE concentrates on communication of information between computer applications. TOVE researchers are studying the problems associated with the use of different representation languages and are investigating the requirements for a "language in which enterprise knowledge can be expressed."

Fox (1992) states that a representation supports integration if it can be used directly, or transformed so that its content can be used, by existing analysis and support tools of the enterprise. They propose the following steps toward EI:

1.   Create a shared ontology (see the section on the IICE Ontology Thrust below) for the enterprise that each agent in the enterprise can use and understand jointly.

2.   Define the meaning of each term in the ontology.

3.   Implement the semantics of the terms in a set of axioms, enabling common-sense based deduction to answer questions about the enterprise.

4.   Define a term and concept symbology.

Heterogeneous Database Integration

Three approaches to heterogeneous database (DB) integration are the composite approach, the federated approach, and CARNOT.

- **Composite Approach:** The composite approach uses a global schema — that is, a large DB schema encompassing all the schemas for a given collection of DBs — to describe the information in the DBs to be integrated. It therefore provides the illusion of a central DB. Semantic conflicts are taken care of *in advance*. The problems with this approach include the fact that a new global schema must be constructed every time a local schema is added or modified.

- **Federated Approach:** The federated approach supports using a collection of schemas for individual DBs together with tools to support information sharing. The users are responsible for resolving any inconsistencies. The problem with this approach is that there is no global schema to provide advice about semantics; thus each DB must maintain some knowledge about other participating DBs. This implies possible computational problems. However, the advantage of this approach is that it is easier to maintain and to deal with inconsistencies. Furthermore, security management is easier than with the other approaches.

- **CARNOT:** CARNOT is an attempt to implement the composite approach using the CYC[49] knowledge base as a global schema.

Method Integration

Mayer et al. (1992) characterize two different aspects: intra- and inter-method integration. The two characteristics of method integration are (1) interoperability (the method application activities are coordinated), which takes advantage of procedural similarities between the methods to achieve this characteristic; and (2) data sharing, which takes advantage of overlap between method concepts to achieve data sharing.

Mayer et al. (1992) also differentiate between two kinds of integration: (1) physical integration, which only requires simple sharing of data and procedures; and (2) logical integration, which requires more complex sharing. A set of methods is considered logically integrated if indirect relations between non-overlapping concepts among pairs of methods have been identified.

Method integration allows for data sharing and change propagation. The two levels of integration, physical and logical, determine the extent to which data can be shared and changes propagated.

Information Model Integration

Huhns (1992) identifies the following needs for information model integration.

1. Need for a coherent picture of the enterprise. Typically, a model describes a portion (physical or logical) of the enterprise, some aspect of a business environment (DBs, DB applications, knowledge bases, business work flow, business information, business resources, and business organization). The integration of a set of models would provide a global view of the enterprise.

2. Need for managing interoperation between portions of an enterprise (e. g., producing new composite information from models of different, or possibly the same, portions of an enterprise or propagating change).

3. Need for the means to validate models in a global context.

---

[49] This is a large, well known project attempting, roughly, to encode an encyclopedia into an intelligent software agent to enable it to reason in a manner similar to human agents.

4.   Need for detecting inconsistencies among the portions of the enterprise being modeled in several different models.

Integration of Knowledge Base and Database Technologies

Neches et al. (1991) identify the following needs for knowledge base and database integration.

- Access to large amount of existing shared data for knowledge processing.

- Efficient management of data and knowledge (in particular, provide knowledge bases with persistent storage capability).

- Intelligent processing of data, in particular the ability to identify redundancy, ambiguity, and inconsistency across knowledge bases and databases.

Computer-Integrated Manufacturing Integration

Computer Integrated Manufacturing — Open System Architecture (CIMOSA) (Kosanke, 1992) is an architecture for the definition, specification, and implementation of computer-integrated manufacturing (CIM) systems. CIMOSA relies heavily on the use of enterprise models. Enterprise models control and monitor the operation of the enterprise. Modeling methods (such as ER, Yourdan, the IDEF suite, and so on) provide for evolution in enterprise model building and correction of existing models. CIMOSA provides an architecture to create and maintain the enterprise models, and a set of services which support model generation and execution.

CIMOSA allows an enterprise to be modeled as a set of loosely coupled processes sharing common enterprise objects and communicating with each other by exchanging object views and events. It also provides for the description of lower levels of functionality and related dynamic behaviors.

The result of applying CIMOSA is a complete description (model) of the enterprise. This description is an executable network of enterprise activities connected through procedural rules which describe the control flow. The connectivity defines the possible interaction between pairs of elements in terms of information and messages sent and received by the elements.

CIMOSA distinguishes between passive and active components. Active components (also called functional entities) include humans, devices, and applications. Passive components include data and information.

- Functional entities, data and information, and functions all need to be integrated.

- Data and information integration is achieved when the right information can be used at the right place and the right time whenever it is stored in the system under any format.

291

- Functions include processes and activities. Function integration is achieved when a function can be executed cooperatively or concurrently in a computer-controlled environment.

Finally, CIMOSA categorizes enterprise modeling in the following way. Each model can be made at three levels: requirements, design, and implementation. At each level, the model can be accessed through different views. CIMOSA proposes four such views: the function view, the information view, the resource view, and the organization view.

Approaches to Enterprise Integration

Shared Dependency Engineering

The goal of shared dependency engineering (SHADE) is to provide a framework for information sharing and communication among the members (people and computer applications) of a product development team (Tenenbaum et al., 1992). To achieve this goal, SHADE uses a shared knowledge base (SKB) with KIF as a representation language. The SKB representation is based on a common ontology that contains all the concepts (classes, relations, functions, etc.) of design knowledge.

Knowledge sharing among participating tools is made possible using *content-based information routing*. The components of content-based information routing are *publication, subscription and notification*, and *relevance determination*. Publication is the mechanism used by the tools to export information to the SKB. A publication is a set of KIF sentences. To be *notified* of changes or new information published by other tools, a tool may *subscribe* to some information. A subscription is a query expressed in a KIF sentence. When new information is published by one of the participating tools, SHADE uses the subscriptions made by the other tools to determine where to send the corresponding notification. Finally, relevance determination is concerned with the problem of deciding what information is relevant to a given subscription. Therefore, it is used to determine when notifications should be sent to the tools, based on their respective subscriptions.

The SHADE project seems to be concerned mainly with the design of SKBs and the routing of information.

Palo Alto Collaborative Test-Bed

The Palo Alto Collaborative Test-Bed (PACT) serves as a test-bed for solving concurrent engineering problems using a knowledge-sharing approach (Cutkosky et al., 1993). Its objective is to create a representational framework in which the component descriptions that make up a design are organized dynamically in a coherent manner. PACT's approach is to experiment with an original framework architecture to uncover the deeper problems and issues involved in solving the concurrent engineering problems.

The framework is composed of agents (computer applications or systems) and facilitators that allow the agents to communicate. This architecture is called a "federation architecture." The logical view of the framework is that the tools interact with respect to a "shared model." In the implementation of this view, the tools have associated local knowledge base reasoners and communicate by sending messages to their local facilitators. The messages sent by the agents must adhere to given protocols. The facilitators are responsible for the support of these protocols, in particular, message passing between the agents, the routing of outgoing messages, the translation of incoming messages, and the initialization and monitoring of agents.

One novelty of the PACT approach is that the tools are not standardized or unified but are encapsulated and use a "shared language of engineering" to communicate. However, early experiments have shown that PACT must still face some important challenges. As it stands now, the framework does not use a common ontology; rather, it relies on the agents' agreement on the format, language, and vocabulary to be used. PACT also does not support different abstraction and granularity in the participating tools. The agents must communicate at the same level of granularity. Finally, the propagation mechanism needs to be refined to improve efficiency and avoid bottlenecks.

Neches' Architecture for an Integrated System

Neches et al. (1991) describe their vision of the organization of an individual knowledge-based application system participating in the KSE. Their system is composed of three main elements:

- A set of services operating on combined (or indistinguishable) knowledge bases. A service can be a generic reasoning module such as an inference mechanism or a task-level reasoning module such as a diagnoser. Other services that may be included in the set are more conventional modules such as spreadsheet, hypertext linkers, electronic mail, and so forth. The services in the set typically would be acquired off the shelf and, possibly, tailored to the specific application domain.

- A uniform user interface manager. The manager is responsible for mediating interactions with humans.

- A set of agents responsible for mediating information with other systems. The presence of such agents implies the usage of a communication language. The language proposed here is called Knowledge Query and Manipulation Language (KQML).

In addition to these three main elements, the system may also contain a shared ontology.

MCC's Enterprise Integration Solution Architecture

MCC's EI solution has five high-level features: execution environment, application architecture, namespace, enterprise characterization, and federation mechanism.

- Execution Environment (EE). EE, which refers to the system's distributed computing environment, EE is composed of basic computing services for which various EE architectures exist. A key issue is the interoperability between the different architectures.

- Application Architecture (AA). An AA is a model of the use of services and tools in a given environment. An AA contains applications and tools for accomplishing specific tasks. The idea is that the binding between a particular AA and different EE platforms should be done at run time. This is possible if each EE can register services in a given, navigable space, and the AA can query and browse that space.

- Namespace (NS). The NS is the space that supports run-time binding between AAs and EEs. It contains, among other things, a description of the services provided by the Ees and the services' protocols. Defining the characteristic of the NS is a key issue in an EI solution. The notion and use of models will have to become predominant in any EI solution. The models will be used to describe behavior, data, and processes. They will be used for manipulating the enterprise through the computing infrastructure. Models must then be bound to AAs. This binding will also be done through an NS.

- Enterprise Characterization (EC). Many parts of the enterprise will need to be characterized (the EE services in particular). EC will include characterization of services and of work processes, product data, physical plant, and so forth. EC will be a key feature of any EI solution because every component of such a solution will require some form of characterization. An EC can be seen as a set of models characterizing the behavior of the enterprise.

- Federation Mechanism (FM). An FM is a mechanism through which heterogeneous elements can be related and connected. The elements may be models, AAs, or Ees. An example of a model FM is a collection of meta-models together with their maintenance, compilation, and linking mechanisms. An FM operates at a high-level of abstraction (i.e., it deals with concepts, knowledge, and information)

APPENDIX D
THE NEUTRAL INFORMATION
REPRESENTATION SCHEME (NIRS)

First-Order Languages

Basic Vocabulary

The basic vocabulary of a first-order language includes several kinds of symbols:

- constants

- variables

- function symbols

- predicates

- logical symbols

Constants correspond to names in ordinary language. For many purposes, it is useful to use abbreviations of names straight out of ordinary language for constants (e.g., j for John, wp for Wright-Patterson, v for Venus, o for Ohio, etc.). When we are describing languages in general and have no specific application in mind, we will simply use the letters a, b, c, and d, perhaps with subscripts; we will assume that we will add no more than finitely many subscripted constants to our language.[50] Constants are usually lowercase letters, with or without subscripts, but this is not necessary. Indeed, it is often useful to use uppercase.

We will often want to say things about an "arbitrary" constant as a way of talking about all constants, much as one might talk about an arbitrary triangle ABC in geometry as a way of proving something about all triangles in general. For this purpose, it will not do to talk specifically about a given constant (for example, a) because we want what we say to apply to *all* constants generally. This requires that, when we are talking *about* our language, we use special *metavariables* whose roles are to serve as placeholders for arbitrary constants of our language, much as "ABC" above serves as a placeholder for arbitrary triangles. Thus, metavariables are not themselves part of our first-order language L but rather part of the extended English we are using to talk about the constants that *are* in the language. We will use the lowercase *sans serif* characters a, b, c for this purpose.

---

[50] The restriction to a finite number of constants here is not essential, but constraint languages in general will use only finitely many; the same holds for predicates and function names below.

Next, are the variables, whose purpose will be clarified in detail below. The lower case letters x, y, and z, possibly with subscripts, will play this role, and we will suppose there to be an unlimited store of them. We will use the characters x, y, and z as metavariables over the store of variables in our language.

Third, are function symbols which correspond most closely in natural language to expressions of the form "The X of", where X is a common noun phrase like "color", "yearly salary", "mother", and so forth, or expressions of the form "The Y-est X in", where Y is an adjective like "smart", or "mean" and X once again by a common noun phrase. Now, common noun phrases typically express general properties. For any common noun phrase CNP, the result of replacing X with CNP in either of the above forms (together with an adjective for Y in the second form) intuitively names a function f that, when applied to a given object a, yields the appropriate instance $f(a)$ of the property expressed by the CNP for that object. Thus, where X is "color", the resulting function in the first form yields the color of the object it is applied to; where it is "yearly salary", the resulting function yields an appropriate dollar amount. Similarly, "The smartest woman in" expresses a function that takes places (e.g., cities, universities, etc.) and yields for each such place the smartest woman therein.

For the most part we will confine our attention to "one-place" functions such as those above that take a single object to another object. But as we will see, there are occasions when we will want to represent functions of more than one argument as well. Examples of expressions that stand for two-place functions are "The only child of ... and ..." and "The sum of ... and ...". Intuitively, the former expresses a partial function[51] from couples with a single child to that child, and the latter simply expresses the addition function, which takes two given numbers to a further number, viz., their sum.

As with constants, in practice it is often convenient just to abbreviate relevant ordinary language functional expressions in defining the function symbols of a formal language. Once again, though, for general purposes, we will just use the letters f, g, and h, possibly with subscripts, for our basic function symbols, and corresponding *sans serif* characters as metavariables. Function symbols designed to stand for functions of more than one argument will be indicated with an appropriate numerical superscript. As above, we will suppose there are only finitely many of these symbols in our language.

We also introduce the symbol • and stipulate that, where g stands for any n-place function symbol in our language and f stands for any one-place function symbol, f • g is an n-place function symbol as well. This

---

[51] That is, a function that might not be defined on every element of its domain. For example, the square root function is only a partial function on the natural numbers because it isn't defined on those numbers which are not squares of other numbers. The function in the text here is partial because its intuitive domain is the set of pairs of humans, and not every such pair has a single child.

corresponds in ordinary language to the fact that we can nest functional expressions (e.g., "The salary of the father of the smartest woman in largest university in ...", or "The successor of the sum of ... and ...").

The fourth group of symbols in our language consists of n-place predicates, $n > 1$. One-place predicates correspond roughly to verb phrases like "is a computer scientist", "has insomnia", "is an employee", and so forth, all of which express properties. Two-place predicates correspond roughly to transitive verbs like "loves", "is an element of", "is less than", "begat", and "lives with", which express two-place *relations* between things. There are also three-place relations, such as those expressed by "gives" and "between"; with a little work we could come up with relations of more than three places, but in practice we shall have little cause to go much beyond this.

We will use uppercase roman letters such as P, Q, and R for predicates, and again corresponding *sans serif* characters as metavariables over predicates. Occasionally predicates will appear with numerical superscripts to indicate the number of places of the relation they represent and, if necessary, with subscripts to distinguish those with the same superscripts. Once again, though, in practice it is often useful to abbreviate relevant natural language expressions. Most languages contain a distinguished predicate for the two-place relation *is identical to*. We will use the symbol $\approx$ for this purpose.

To drive home the difference at this point between predicates and function symbols, note that a function symbol combines with names to yield yet another name-like (i.e., referring) expression (e.g., to draw on ordinary language, the function symbol "the husband of" combines with the name "Di" to yield the new referring expression — or *definite description*, as such are often called — "the husband of Di". On the other hand, a (one-place) predicate combines with a name to form a *sentence*, something that can be true or false, not a name-like expression. Thus, the predicate expression "is happy" combines with the name "Di" to yield the sentence "Di is happy". The same is easily seen to hold for n-place predicates generally.

The last group of symbols consists of the basic logical symbols: $\neg$, $\wedge$, $\vee$, $\supset$, $\equiv$ the existential quantifier $\exists$, and the universal quantifier $\forall$. We will also need parentheses and perhaps other grouping indicators to prevent ambiguity.

297

Grammar

Now that we have our basic symbols, we need to know how to combine them into grammatical units or *well-formed formulas*, the formal correlates of sentences. These will be the expressions that can encode the sort of information we will want to express in our theory (and more besides). This is done *recursively* as follows.[52]

First, we want to group all name-like objects into a single category known as *terms*. This group will of course include the constants and, for reasons cited below, the variables as well. However, recall the discussion of function symbols above. There we saw that an expression like "The yearly salary of" seems to name a function on objects; but the values of functions are objects as well. Therefore, when we attach a namelike "Fred" to the functional expression above, the result — "The yearly salary of Fred" — is a sort of name for Fred's yearly salary. Thus, we count the result of attaching a functional symbol to an appropriate number of constants and/or variables as a term as well; such terms can also be among the terms to which a function symbol attaches. Thus, more exactly, letting t_1, t_2, ... stand for arbitrary terms and f stand for an arbitrary function symbol, if t_1, ..., t_n are terms and f is an *n*-placefunction symbol, $f(t_1, ..., t_n)$ is a term as well.

Terms formed out of certain familiar two-place function symbols (examples of which will be introduced below) are more commonly written in *infix* notation rather than the prefix notation just defined, with the function symbol flanked by the two terms rather than preceding them. Thus, for a two-place function symbol f and terms t, t′, the term f(t, t′) can also be written as tft′. So, for example, +(2,3) can be written as 2+3.

Next we define the basic formulas of our language. Just as verb phrases and transitive verbs in ordinary language combine with names to form sentences, so in our formal language predicates combine with terms to form formulas. Specifically, if P is any *n*-place predicate, and t_1, ..., t_n are any *n* terms, then $Pt_1 ... t_n$ is a formula — in particular an *atomic* formula. To illustrate, if *H* abbreviates the verb phrase "is happy", and *a* the name "Annie", then the formula *Ha* expresses the proposition that Annie is happy. Again, if *L* abbreviates the verb "loves", *b* the name "Bob", *c* the name "Charlie", and *f* the expression "the fiancee of", then the formula *Lbf(c)* expresses the proposition that Bob loves Charlie's fiancee.

Often when one is using more elaborate predicates drawn from natural language (e.g., if we had used *LOVES* instead of *L* in the previous example), it is more readable to use parentheses around the terms in atomic formulas that use the predicate and separate them by commas (e.g., *LOVES(b,x)* instead of *LOVESbx*). Thus, more

---

[52] That is, the definition is given in such a way that complex cases of the class being defined are defined in terms of simpler cases of the same class. Thus, recursive definitions often look circular, but they are not because they always begin with well-grounded initial cases not defined in terms of other members of the class being defined.

generally, any atomic formula $Pt_1 \dots t_n$ also can be written as $P(t_1,\dots,t_n)$. Furthermore, atomic formulas involving some familiar two-place predicates like $\approx$, and a few others that will be introduced below, are more often written using infix rather than prefix notation. For example, we usually express that a is identical to b by writing $a \approx b$ rather than $\approx ab$. Thus, we stipulate that formulas of the form Ptt' can also be written as tPt'.

Now we begin introducing the logical symbols that allow us to construct more complex formulas. Intuitively, the symbol $\neg$ expresses negation; that is, it stands for the phrase "It is not the case that". Since we can negate any declarative sentence by attaching this phrase onto the front of it, we have the corresponding rule in our formal grammar that if φ is any formula, then so is $\neg\varphi$. The symbols $\wedge$, $\vee$, $\supset$, and $\equiv$ stand roughly for "and", "or", "if ... then", and "if and only if", which are also (among other things) operators that form new sentences out of old in the obvious ways. Unlike negation, though, each takes *two* sentences and forms a new sentence from them. Thus, we have the corresponding rule that if φ and ψ are any two formulas of our language, then so are $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \supset \psi)$, and $(\varphi \equiv \psi)$.

Finally, we turn to the quantifiers $\exists$ and $\forall$. Recall that we introduced variables without explanation above. Intuitively, $\exists$ and $\forall$ stand for "some" and "every", respectively; the job of the variables is to enable them to play this role in our formal language. Consider the difference between "Annie is happy", "Some individual is happy", and "Every individual is happy". In the first case, a specific individual is picked out by the name "Annie" and the property of being happy is predicated of her. In the second, all that is stated is that some unspecified individual or other has this property. In the third, it is stated that every individual, whether specifiable or not, has this property. This lack of specificity in the latter two cases can be made explicit by rephrasing them as follows: for some (respectively,, every) individual x, x is happy. Since the rule for building atomic formulas counted variables among the terms, we have the means for representing these paraphrases. Let H abbreviate "is happy" once again; then we can represent the paraphrases as $\exists x Hx$ and $\forall x Hx$, respectively.

Accordingly, we add the final rule to our grammar: if φ is any formula of our language and v is any variable, then $\exists v \varphi$ and $\forall v \varphi$ are formulas as well. In such a case, we say that the variable v is *bound* by the quantifier $\exists$ (resp., $\forall$), and we say that the formula φ is the *scope* of the quantifier $\exists$ in $\exists v \varphi$, and it is the scope of the quantifier $\forall$ in $\forall v \varphi$.

First-Order Semantics

Structures and Interpretations

We have motivated the construction of our grammar by referring to the intended meanings of the logical symbols and by letting our constants and variables abbreviate meaningful expressions out of ordinary language. However, from a purely formal point of view, all we have in a language is a bunch of uninterpreted syntax; we

haven't described in any formal way how to assign meaning to the elements of a first-order language. We will do so now.

A *structure* for a first-order language L consists simply of two elements: a set D called the *domain* of the structure and a function Φ known as an *interpretation function* for L. Intuitively, D is the set of things one is describing with the resources of L (e.g., the natural numbers, major league baseball teams, the people and objects that make up an air force base, or the records inside a database). The purpose of Φ is to fix the meanings of the basic elements of L in terms of objects in or constructed from D.

Interpretations of Constants and Function Symbols

The interpretation function works as follows. First, we deal with terms. We begin by noting that variables will not receive an interpretation because their meanings can vary (they are *variables* after all) within a structure. They will be treated with their own special semantic apparatus below. On the other hand, constants, being the formal analogs of names with fixed meanings, are assigned members of D once and for all as their interpretation; in symbols, for all constants κ of L, $\Phi(\kappa) \in D$.

To deal with terms formed from function symbols, we first need to interpret the function symbols themselves. To begin with, each basic function symbol α is assigned a function $\Phi(\alpha)$ from D into D. Intuitively, as indicated above, the functions expressed in ordinary language are often *partial*; that is, often they are not defined everywhere. For example, the function expressed by "The salary of" is not defined when applied to a conveyer belt or a garden vegetable. This suggests that we ought to let the functions from D into D that interpret our function symbols be partial. However, this leads to certain inelegancies in our formal apparatus; therefore, we opt instead to include a distinguished object $\perp$ in our domain D whose sole purpose is to be the value of functions applied to objects on which they are intuitively undefined. Thus, if we have a function symbol f abbreviating "The salary of", and if our domain D contains both persons and conveyer belts, then the interpretation of f will be the function that takes each person to his or her salary in dollars and every kind of other object to our distinguished object $\perp$. Formally, then, for all basic *n*-place function symbols a of L, $\Phi(\alpha) \in \{\Phi \mid \Phi : D^n \to D\}$; that is, the interpretation of a basic *n*-place function symbol a of L is going to be an element of the set of all *n*-place functions from the set of *n*-tuples of the domain D into D.

We need to address the nonbasic function symbols — that is, those of the form $\alpha \cdot \beta$ which correspond to nested functional expressions in ordinary language like "The salary of the father of". Intuitively, we want $\Phi(\alpha \cdot \beta)$ to be the composition of $\Phi(\beta)$ with $\Phi(\alpha)$, that is, $\Phi(\alpha) \circ \Phi(\beta)$, where in general $(\Phi \circ \Psi)(x) = \Phi(\Psi(x))$.[53]

Interpretations of Predicates

Finally, for any one-place predicate P, we let $\Phi(P)$ be a subset of D — intuitively, the set of things that have the property expressed by P. For any n-place predicate R, $n > 1$, we let $\Phi(R)$ be a set of n-tuples of elements of D — intuitively, the set of n-tuples of objects in D that stand in the relation expressed by R. Thus, for example, if we want L to abbreviate the verb "loves", and if our domain D consists of the population of Texas, $\Phi(L)$ will be the set of all pairs $\langle a, b \rangle$ such that a loves b. Thus, formally, for all n-place predicates P, $\Phi(P) \subseteq D^n$.[54]

If one wishes to include the identity predicate $\approx$ in one's language and have it carry its intended meaning, one needs an additional, more specific semantical rule designed to do this. Identity, of course, is a relation that holds between any object and itself, but not between itself and any other object. This additional semantical constraint is easy to express formally: if our language L contains $\approx$, then the interpretation of $\approx$ is the set of all pairs $\langle o, o \rangle$ such that o is an element of the domain D (i.e., more formally, $\Phi(\approx) = \{\langle o, o \rangle \mid o \in D\}$).

Truth

Variable Assignments

Given a structure $\mathbf{M} = \langle D, \Phi \rangle$ for L, we can define what it is for a formula of L to be *true* in $\mathbf{M}$. As usual, this is done recursively. First, we need to introduce the notion of an *assignment* α for the variables, which is a sort of addendum to our interpretation function: it assigns members of the domain to variables. Relative to an assignment function α, we can define the interpretation of a complex term $f(t_1,...,t_n)$ for any function symbol f and any terms $t_1,...,t_n$. An interpretation function F alone does not suffice for this because complex functional terms might contain variables — e.g., the term f(x) — which are ignored by interpretation functions. However, if we

---

[53] Note that _ is a *metalinguistic* symbol of our extended English that expresses the meaning of our object language symbol $\cdot$, viz., the composition function.}—in terms of our example, the composition of the function expressed by "The salary of" with the function expressed by "the father of." Notice that by our trick with $\wedge$, the composition of any two functions will always be total.

[54] Where $D^1 = D$, and $D^{n+1} = D^n \times D$; that is, D^1 is just D itself, D^2 is the set of all pairs of members (i.e., the Cartesian product $D \times D$) of D, D^3 the set of all triples of members of D, and in general D^n is the set of all n-tuples of members of D.

supplement Φ with an assignment α for the variables, we have something for the function F(f) to work on. Specifically, the interpretation of the term f(x) *under* α, Fa(f(x)), is just the function F(f) applied to a(x), the value assigned to x by a.

In general, let Φ_α be the result of adding α to Φ.[55] Then the interpretation $\Phi_\alpha(f(t_1,...,t_n))$ of a complex term $f(t_1,...,t_n)$ under a is simply the result of applying the function $\Phi_\alpha(f)$ (which is just $\Phi(f)$, since f is a function symbol) to the objects $\Phi_\alpha(t_1),...,\Phi_\alpha(t_n)$, i.e., $\Phi_\alpha(f)(\Phi_\alpha(t_1),..., \Phi_\alpha(t_n))$.

Truth Under an Assignment

Atomic Formulas. Our goal in this section is to define the notion of a formula being *true* in a structure **M**. To do so, we will first define a closely related notion, viz., that of truth *under an assignment* α. For convenience, we will sometimes speak of a formula's being "true$_\alpha$ in **M**" instead of being "true in **M** under α." We start by defining this notion for atomic formulas. Let φ be an atomic formula $Pt_1...t_n$. Then φ is true$_\alpha$ in **M** just in case $\langle \Phi_\alpha(t_1),..., \Phi_\alpha(t_n) \rangle \in \Phi_\alpha(P)$. Intuitively, where $n = 1$, Pt is true$_\alpha$ in **M** just in case the object in D that t denotes is in the set of things that have the property expressed by P. For $n > 1$, $Pt_1...t_n$ is true$_\alpha$ just in case the *n*-tuple of objects $\langle o_1,...,o_n \rangle$ denoted by $t_1,...,t_n$ respectively is in the set of n-tuples whose members stand in the relation expressed by P (i.e., just in case those objects stand in that relation).

Let us actually construct a small language **L*** and build a small structure **M*** to illustrate these ideas. Suppose we have four names *a*, *b*, *c*, *d*, a single function symbol *h* (intuitively, to abbreviate "the husband of"), a one-place predicate *H* (intuitively, to abbreviate "is happy"), and a three-place predicate *T* (intuitively, to abbreviate "is talking to ...about"). Let us also include the distinguished predicate $\approx$, though we'll make no real use of it until later. We will use *x, y,* and *z* for our variables.

For our structure **M***, we will take our domain D to be a set of three individuals, {Beth, Charlie, Di}, and our interpretation function G will be defined as follows. For our constants, $\Gamma(a) = \Gamma(b) =$ Beth, $\Gamma(c) =$ Charlie, and $\Gamma(d) =$ Di. (Beth has two names in our language in order to illustrate a point to be made several sections hence.) For our function symbol *h*, we let $\Gamma(h)$(Beth) = $\Gamma(h)$(Charlie) = $\perp$ (so that $\Gamma(h)$ is "undefined" on Beth and Charlie), and $\Gamma(h)$(Di) = Charlie. For our predicates *H* and *T*, we let $\Gamma(H)$ = {Beth, Di} (so, intuitively, Beth and Di are happy), and $\Gamma(T)$ = { $\langle$Beth, Di, Charlie$\rangle$, $\langle$Charlie, Charlie, Di$\rangle$} (so, intuitively, Beth is talking to Di about Charlie, and Charlie is talking to himself about Di). Following the rule for $\approx$, we let $\Gamma (\approx) = \langle$Beth, Beth$\rangle$, $\langle$Charlie, Charlie$\rangle$, $\langle$Di, Di$\rangle$}. Finally, for our assignment function β, let $\beta(x) = \beta(y) =$ Charlie and $\beta(z) =$ Di.

---

[55] That is, if x is a constant, function symbol, or predicate, Fa(x) = F(x); if x is a variable, then Fa(x) = a(x).

Let's now check that *Hd* and *Tbdh(z)* are true in **M*** under β. In the first case, by the above, *Hd* is true$_β$ in **M*** just in case $Γ_β(d) ∈ Γ_β(H)$ — that is, just in case Di is an element of the set {Beth, Di}, which she is. So *Hd* is true$_β$ in **M***. Similarly, *Tbdh(z)* is true$_β$ in **M*** just in case $⟨Γ_β(b), Γ_β(d), Γ_β(h(z))⟩ ∈ Γ_β(T)$, that is, just in case $⟨Γ(b),$ $Γ(d), Γ(h)(β(z))⟩ ∈ Γ(T)$, that is, just in case $⟨$Beth, Di, $Γ(h)$(Di)$⟩ ∈$ {⟨Beth, Di, Charlie⟩, ⟨Charlie, Charlie, Di⟩}, that is, just in case ⟨Beth, Di, Charlie⟩ $∈$ {⟨Beth, Di, Charlie⟩, ⟨Charlie, Charlie, Di⟩. Since this obviously holds, the formula *Tbdh(z)* is true$_β$ in **M***.

A formula is *false$_α$* in a structure **M**, of course, just in case it is not true$_α$ in **M**. It is easy to verify that, for example, *Hh(b)*, *Hx*, and *Tdbc* are all false$_β$ in **M*** under β.

Conjunctions, Negations, etc. Now for the more complex cases. First, suppose that φ is a formula of the form ¬ψ. Then φ is true$_α$ in a structure **M** just in case ψ is *not* true$_α$ in **M**. In so defining truth for negated formulas we ensure that the symbol ¬ means what we've intended. Things are much the same for the other symbols. Thus, suppose φ is a formula of the form ψ ∧ θ. Then φ is true$_α$ in **M** just in case both ψ *and* θ are. If φ is a formula of the form ψ ∨ θ, then φ is true$_α$ in **M** just in case either ψ *or* θ is. If φ is a formula of the form ψ ⊃ θ, then φ is true$_α$ in **M** just in case either ψ is false$_α$ in **M** or θ is true$_α$ in **M**. If φ is a formula of the form ψ ≡ θ, then φ is true$_α$ in **M** just in case ψ and θ have the same truth value in **M**.

The reader should test his or her comprehension of these rules by verifying that ¬*Hh(b)* and (*Tbdh(z)* ∧ *Tccy*) ⊃ *Hd* are both true in **M*** under β.

Quantified Formulas. Lastly, we turn to quantified formulas. The intuitive idea is this: When we introduced the quantifiers above, we noted that "Some individual is happy;" that is, ∃*xHx*, can be paraphrased as "for some value of the variable '*x*,' the expression '*x* is happy' is true." This is essentially what our formal semantics for existentially quantified formulas will come to. To anticipate things a bit, ∃*xHx* will be true in a structure **M** under α, roughly, just in case the *un*quantified formula *Hx* is true in **M** under some (in general, new) assignment α′ such that α′(*x*) is in the interpretation of *H*. It is easy to verify that this formula is true in our little structure **M*** under β, when we look at a new assignment function β′ that assigns either Beth or Di to the variable *x*. Thus, ∃*xHx* should come out true in **M*** under β.

But we have to be a little more careful because some formulas — *Tcxz*, for example — contain more than one unquantified variable. Thus, when we are evaluating a quantification of such a formula — such as $zTcxz — have to be sure that the new assignment function α′ doesn't change the value of any of the unquantified variables (in this case, the variable *x*). Otherwise, we could change the sense of the unquantified formula in mid-evaluation. Intuitively, under the assignment function β above, ∃*zTcxz* states that Charlie is talking to himself about someone (recall that β(*x*) = Charlie); this statement should turn out to be true$_β$ in **M*** because Charlie is talking to himself about Di (i.e., ⟨Charlie, Charlie, Di⟩ $∈ Γ_β(T)$). However, suppose all we require is that there be some new assignment function β′ such that β′(*z*) is Di. Then it could turn out also that β′(*x*) is Beth; but the formula *Tcxz*

would not be true in **M*** under β because Charlie is not talking to Beth about Di (i.e., ⟨Charlie, Beth, Di⟩ ∉ Γ_β (*T*)), and hence we would not be able to count ∃*zTcxz* as true in **M*** under β after all as we should like.

All that is needed is a simple and obvious restriction: when evaluating the formula ∃*zTcxz*, the new assignment function that we use to evaluate *Tcxz* must not be allowed to differ from β on any variable except *z* (and even then it *needn't* differ from β, in which case it just *is* β). More generally, we put the matter like this: if φ is an existentially quantified formula ∃ν*ψ*, then φ is true in a structure **M** under α just in case there is an assignment function a′ just like α except perhaps in what it assigns to ν such that the formula ψ is true in **M** under a′. Furthermore, if φ is a universally quantified formula ∀ν*ψ*, then φ is true in **M** under a just in case for *every* assignment function a′ just like a except perhaps in what it assigns to ν the formula ψ is true in **M** under a′. That is, in essence, ψ is true in **M** just in case ψ is true in **M**, no matter what value in the domain we assign to ν (while keeping all other variable assignments fixed).

The reader can once again test his or her comprehension by showing in detail that ∃*xTxbh(z)* is false in **M*** under β and that ∀*x(Hx* ∨ *Tbdx)* is true in **M*** under β.

Truth

Finally, we can define a formula to be *true* in a structure **M** *simpliciter* just in case it is true_α in **M** for all assignments α, and *false* in **M** just in case it is false_α in **M** for all α. Note, on this definition that, for almost any interpretation, there will be formulas that are neither true nor false in the interpretation. Our example ∃*zTbxz* above, for instance, is neither true nor false in **M*** because there are assignments α on which it comes out true_α — all those on which α(*x*) = Di — and assignments α on which it comes out false_α — all those on which α(*x*) ≠ Di. Such formulas will always have free variables because it is the semantic indeterminacy of such variables that is responsible for this fact. However, note that some formulas with free variables will be true or false in some models, though these typically will be logical truths (respectively, falsehoods) like *Hx* ∨ ¬*Hx* (i.e., formulas that are true (respectively, false) on every interpretation).

First-Order Logic

Propositional Logic

Now that we've got the notion of a first-order language and its semantics, we want to capture the meanings of the logical constants ¬, ∧, ∨, …, ≡, ∀, and ∃ as explicated in the semantics. We will do this in the usual way by developing a rigorous and precise *logic*. A logic, in the sense relevant here, is a systematic characterization of correct principles of reasoning with respect to a given cluster of concepts. The concepts here, of course, are those expressed by the logical constants above, corresponding roughly to the ordinary language concepts of negation (*not*}, or *it is not the case that*), conjunction (*and*), disjunction (*or*), material implication (*if … then*), material

equivalence (*if and only if*), existential quantification (*some*), and universal quantification (*every*, or *all*). The form such a system takes usually consists of two components: *axioms* and *rules of inference*. We start with the axioms for the propositional connectives.

Axioms for Propositional Connectives. The axioms for the propositional connectives $\neg$, $\wedge$, $\vee$, $\supset$, and $\equiv$ constitute the basis of propositional logic and can be thought of as characterizing their meanings. There are many equivalent axiomatizations for propositional logic, but the following, which makes use of the notion of an axiom schema, is one of the easiest. An axiom schema is not itself an axiom but rather a sort of template — a general form any instance of which is an axiom. Thus, axiom schemas are not themselves part of the language. That is, where j, y, and q are formulas, any instance of any of the following schemas is an axiom:

A1 $\varphi \supset (\psi \supset \varphi)$

A2 $(\varphi \supset (\psi \supset \theta)) \supset ((\varphi \supset \psi) \supset (\varphi \supset \theta))$

A3 $(\varphi \supset \psi) \supset ((\Re \varphi \supset \psi) \supset \psi)$

In English, **A1** says essentially that if a sentence φ is true, then for any other sentence ψ whatever; if ψ is true, then φ is still true. **A2** says that if a sentence φ implies that if ψ is true then so is θ, then if φ implies ψ, it also implies θ. Finally, **A3** says essentially that if a sentence φ implies another sentence ψ, then if ψ is also implied by the negation of φ, ψ is true no matter what (since either φ or its negation is true no matter what). These axioms are pretty much as trivial as they sound. But, like the elementary truths of arithmetic or geometry that are second nature to us, they must be stated explicitly as a basis for deriving other, less obvious truths; they cannot be conjured out of thin air.

Notice that axiom schemas only use the two connectives $\neg$ and $\supset$. Even though we've been using the other propositional connectives all along, officially we will consider these to be our two "primitive" connectives; the others can be defined in terms of them as follows (where the symbol $=_{df}$ means "is defined as"):

Def 1: $(\varphi \vee \psi) =_{df} (\Re \varphi \supset \psi)$

Def 2: $(\varphi \wedge \psi =_{df} \neg (\neg \varphi \vee \neg \psi)$

Def 3: $(\varphi \equiv \psi) =_{df} (\varphi \supset \psi) \wedge (\psi \supset \varphi)$

The reader can again test comprehension by showing that, no matter what truth values are assigned to φ, ψ, and θ, the two sides of each definition will always have the same truth values when evaluated in accord with the semantical rules given above for the connectives given above.

Rules of Inference: Modus Ponens

A logic isn't much good without rules of inference; that is, rules that allow us to move from statements we know or assume to be true at the outset (e.g., our axioms) to new statements that follow logically from them (e.g., *to theorems*). Without them, all we could do is write down axioms; there would be no way to infer new truths from those already given. There is only one rule of inference in propositional logic: Modus Ponens.

Modus Ponens (MP): If the formulas φ and $\varphi \supset \psi$ follow from the axioms of propositional logic, we may infer that ψ does as well.

Given this, the notion of theoremhood can be defined precisely as follows. A formula φ is a theorem of propositional logic if and only if there is a sequence $\varphi_1,..., \varphi_n$ such that φ_n is φ and each φ_i is either an axiom or follows from previous lines by **MP**; that is, there are previous formulas $\varphi_j, \varphi_k, j, k < i$, such that φ_k is $\varphi_j \supset \varphi_i$. We can also define the notion of a formula ψ following from a *set* of formulas G in the same way except by adding in addition that ψ_i in the above definition could also be a member of G.

As a simple example using our language L*, consider the following proof of $Hd \supset Hd$ (i.e., the statement *If Di is happy, then Di is happy*. Note that, trivial as it is, $Hd \supset Hd$ is not an instance of an axiom schema; hence, if it is to be a theorem of our system, it must be derivable from the axioms using our rule of inference **MP**. This is in fact the case. As an instance of **A1**, we have:

$Hd \supset ((Hd \supset Hd) \supset Hd)$.

As an instance of **A2** we have:

$(Hd \supset ((Hd \supset Hd) \supset Hd)) \supset ((Hd \supset (Hd \supset Hd)) \supset (Hd \supset Hd))$.

By **MP**, it follows from these two statements that

$(Hd \supset Hd \supset Hd)) \supset (Hd \supset Hd)$.

is an instance of **A1** again, hence by **MP** once more we can infer $Hd \supset Hd$ from the latter two statements.

There are equivalent systems of propositional logic that are more streamlined and computationally more efficient than the basic system here, but this is the foundation on which they are all built and illustrates well enough how the process of deduction works.

Predicate Logic

Axioms for the Quantifiers

When we add axioms for the quantifiers to propositional logic, we have full *predicate* logic, also known as *first-order logic* and *quantification theory*. The quantifiers are interdefinable, so we only need to take one of them as primitive. The axioms for predicate logic are usually stated in terms of the universal quantifier $\forall$, so we will take that as our primitive and shall define $ as follows:

Def 4: $\exists x \varphi =_{df} \neg \forall x \neg \varphi$.

That this definition is intuitively correct is clear on a moment's reflection. So, for example, there exists an x such that x is happy (i.e., someone is happy) if and only if it is not that case that for all x, x is not happy (i.e., if and only if not everyone is unhappy).

We can now state three new quantificational axiom schemas. For any formula φ and term t, we let φ_t^x stand for the result of substituting all unbound occurrences of x in j with t. Then, any instance of the following is an axiom:

A4 $\quad \forall x \varphi \supset \varphi_t^x$ so long as t does not contain, and is not itself, a variable that becomes bound in φ_t^x.

A5 $\quad \forall x(\varphi \supset \psi) \supset (\forall x \varphi \supset \forall x \psi)$.

A6 $\quad \varphi \supset \forall x \varphi$, where x does not occur unbound in φ.

The intuitive idea behind these axioms is straightforward. **A4** simply says that if something is true of everything in general, it is true in particular of anything we can name. Thus, for example, $\forall x\ (NUM(x) \supset \exists y\ (y = x+1)) \supset (NUM(24) \supset \exists y\ (y = 24+1))$; that is, if for every number there is a number one greater than it, then in particular there is a number one greater than 24.

Reverting to our language **L*** and its structure **M***, we have as an instance of this axiom schema:

$\forall x(Hx \vee \exists yTxxy) \supset (Hc \vee \exists yTccy)$.

The antecedent here (i.e., the formula to the left of the $\supset$), $\forall x\ (Hx \vee \exists yTxxy)$, is in fact true in **M***. That is, in **M***, everyone is either happy or talking to themselves about someone in **M***. Thus, if we were to count this as a further "special" axiom — that is, a nonlogical piece of information that characterizes the situation in the specific structure we're investigating and which might well not hold in other structures — we would be able to prove by MP that ($Hc \vee \exists y\ Tccy$); that is, that Charlie is either happy or talking to himself about someone.

The second schema **A5** captures another aspect of the meaning of "every." Consider a simple example: if every individual is such that if it is red then it has a color, then if in fact every individual is red, every individual has a color. This is just an unsymbolized instance of **A5** and illustrates clearly **A5**'s validity.

Finally, **A6** simply says that a quantifier doesn't affect the truth of a formula φ if the quantifier doesn't bind a variable that doesn't occur in φ or — what amounts to the same thing — occurs in φ but is bound by another quantifier. For example, if it is true that Beth is happy, *Hb*, then it is also true for every value of *x* that Beth is happy, ∀*xHb*. Similarly, if Charlie is talking to someone about Di, ∃*zTczd*, it is also true that for every value of *x* Charlie is talking to someone about Di, ∀*x*∃*zTczd*.

Rules of Inference: Generalization

The move to predicate logic with its quantified formulas necessitates a further rule of inference, one designed to capture how we reason with universal quantification. As usual, the idea is best illustrated by an example. Suppose you wanted to prove something about all prime numbers, for example, that for every prime there is a greater prime. You might begin by saying something like, "Let *p* be an arbitrary prime number." You might even pick a specific prime, such as 17. Then, by appealing to none of the specific properties of your chosen prime that distinguish it from other primes (e.g., that it's less than 100, Plato's favorite number, etc.), you proceed to prove in the usual way that there is another prime greater than *p*. You then conclude that the same is true for *every* prime. What permits you to do this is precisely the fact that you did not appeal to any properties of *p* that do not hold for all primes; it was, in a precise sense, arbitrary.

This sort of example illustrates the inference rule known as *Generalization*. Informally, if you can prove that something is true of a particular individual *o* without appealing to anything that couldn't be proved of everything else in the domain, that same thing is true of everything. The way we capture this idea of not appealing to anything that couldn't be proved of everything else is by restricting generalization to formulas whose proofs contain no formulas that say anything about the object being generalized. Thus, we can say that if φ_t^x follows from Γ and the axioms of predicate logic, and *t* does not occur (free) in Γ, then ∀*x*φ follows from Γ and the axioms of predicate logic. Consequently, if t refers to the object *o*, the absence of t from the formulas in Γ indicates they say nothing about *o*. In fact, we can actually use a simpler but equivalent inference rule that only generalizes on variables. This inference rule is:

Gen: If φ follows from the axioms of predicate logic, then ∀xφ does as well.

We noted above that special, or nonlogical, axioms are designed only to hold within a given structure one has singled out (e.g., a structure that models a certain manufacturing or engineering system one might be investigating). A special axiom thus captures the "logic" things within a restricted sphere. However, genuine logical axioms should be exceptionless; a logical axiom formulated within a given language L should be true in all

structures of L. When this property holds of all the axioms of a logical system, the system is said to be *sound*. Soundness is an essential property of any logical system because it is precisely the job of its logical axioms to capture features that hold in any of its structures. Any axiom that was not true in every structure could therefore not rightfully be considered a *logical* axiom and would have to be rejected. It is straightforward (and a good exercise) to show that, for a given language L, any instance of any of the above axiom schemas, and anything provable from them, in fact has this property.[56]

The converse of soundness, that any formula true in every structure follows from the axioms, is known as *completeness* and is much harder to prove. While its absence from a formal system is perhaps not as disastrous as the absence of soundness, completeness is nonetheless a very important and desirable property for a formal system to have because it shows that the semantics and logic of the system match up precisely. It is provable that both propositional and predicate logic are complete.

Identity

Identity and Expressive Power

A very important concept within most any type of formal system is that *of identity* which we will express in our languages by means of the two-place predicate $\approx$.[57] Identity adds a great deal of flexibility and expressive power to a language. Identity is particularly useful in languages that contain function symbols, for with identity one can explicitly identify a named object as the value of a certain function. For example, in our language L*, we can express that Charlie is Di's husband, $c \approx h(d)$.

Second, identity can be used to express the definite article "the." When we ascribe a property to something only identified as "the j" — for example, that *the person Charlie is talking to himself about* is happy — we are implying three things: (1) there is something that fits the description j (i.e., there *is* someone Charlie is talking to himself about), (2) nothing else fits the description (i.e., Charlie isn't talking to himself about anyone else), and (3) j

---

[56] The proof proceeds by ordinary mathematical induction on the number of quantifiers and connectives a formula contains.

[57] We use ª as our identity predicate *within* languages; this is to be distinguished from the concept of identity as it appears in our metalinguistic talk *about* languages and their structures, which we've been expressing with the more familiar =.

has the property in question (i.e., the object of Charlie's attention is happy).[58] All three components are easily expressed in one formula with the help of the identity predicate. Thus, our example here is expressed in our language L* as follows: $\exists x(Tccx \land \neg\exists y\ (Tccy \land x \neq y) \land Hx)$. The force of the "anyone *else*" in (2) above here is captured by the negated identity predicate here in the formula: anyone *other than* (i.e., not *identical to* the person in question).

Finally, similar techniques can be employed to express numerical notions without appealing explicitly to numbers. For example, one can express that *at least two* philosophers are wealthy as $\exists x\exists y(Px \land Py \land \neg x \approx y)$. Note that the third conjunct here is necessary because the bare statement $\exists x\exists y(Px \land Py)$ doesn't imply there are *two* wealthy philosophers — both x and y could be assigned the same unique wealthy philosopher as their values (convince yourself of this by referring to the section on the semantics of $\exists$). In a similar fashion, one can express that there are *exactly two* wealthy philosophers: $\exists x\exists y(Px \land Py \land x \neq y \land \forall z(Pz \supset (z \approx x \lor z \approx y))$ (i.e., in English, there are at least two wealthy philosophers x and y, and any wealthy philosopher is identical with either x or y). Finally, one can also say there are *at most* two wealthy philosophers: $\forall x\forall y\,z\,((Px\ \ Py\ \ Pz)\ \ (x \approx y \lor x \approx z \lor y \approx z))$. Check to see that this statement will be true if there are fewer than three philosophers and false otherwise. These forms are of course also all easily generalizable for any finite number other than two.

Axioms for Identity

Most systems of predicate logic include the notion of identity among the logical constants of the system. Given one standard (though debatable) conception of logic as the study of the *most general* principles of reasoning, this seems quite appropriate because identity is a notion that seems applicable to almost any domain about which one might reason. Regardless of whether identity is a logical notion, it is certainly a notion one might often want to use within a formal system that has been tailored for a certain purpose; in particular, it is essential to our constraint languages. However, the only way to ensure that the identity predicate carries its intended meaning within a given system is to build that meaning into the system by means of appropriate axioms. The usual axioms for identity are, as above, presented in the form of schemas and are straightforward:

A7 $t \approx t$, for any term t

A8 $x \approx t \supset (\varphi \supset \varphi_t^x)$, so long as t does not contain and is not itself a variable that becomes bound in j_t^x

A7 captures the point made above, that identity holds between any object and itself. A8 is nearly as intuitive. The idea is simply that, if something is true of a given object, it doesn't matter how the object is referred

---

[58] This is the essence of Bertrand Russell's *theory of descriptions*, first developed in his famous paper "On Denoting," *Mind* 14 (1905).

to; it's still true of that object.[59] If, for example, Mark Twain wrote *Huckleberry Finn*, it follows that Samuel Clemens did as well because they are the same person. That is, more formally, by **A8** it is an axiom that

$$m \approx s \supset (WROTE(m,h) \supset WROTE(s,h)).$$

Again, if we add $m \approx s$ as a special axiom or derive it from other information we possess, we can prove by **MP** that $WROTE(m,h) \supset WROTE(s, h\})$. In addition, if we then have the further information that $WROTE(m,h)$, we can prove by **MP** that $WROTE(s,h)$.

As a second example, let's revert to our language L* in which we included the identity predicate. In that language, we have both

$$a \approx c \supset (Ha \supset Hc)$$

and

$$a \approx b \supset (Ha \supset Hb)$$

as instances of **A6**. In **M***, $a \approx c$ is false because G(a) = Beth, and G(c) = Charlie. Thus, $a \approx c$ would not be considered among any special axioms we might have to characterize **M***. Hence, as we should hope, we wouldn't be able to infer $Ha \supset Hc$, which is also false in **M***. However, $a \approx b$ is true in **M*** (recall that we assigned both a and b to Beth as their interpretation) and, hence, could be a special axiom for the situation characterized by our structure. By **MP** we could then infer from the second of the two instances above that $Ha \supset Hb$, and from Ha (which might be a further special axiom perhaps) that Hb.

As one would hope, our logic remains sound and complete when we add the axioms for identity.

---

[59] There are well-known exceptions to this axiom. For example, suppose Shorty is five feet tall and his real name is "Eddie." Thus, Shorty ^{a} Eddie. Nonetheless, from the fact that Shorty is so-called because of his size, it doesn't follow that Eddie is so-called because of his size. Other famous contexts where this principle seems to break down are those involving psychological attitudes like belief. For example, even though I believe that 9 is prime, I may not, due to my rusty calculus, believe that $\int_0^3 x^2 dx$ is prime, despite the fact that $\int_0^3 x^2 dx$ ^{a} 9. In the semantics and logic we are constructing, it is assumed that we shall not be needing to formalize expressions like "is so-called because of" and "believes" — though it should be noted that the apparatus we've developed here is eminently capable of being extended to handle such expressions.

Basic Set Theory

Throughout this discussion, we have employed set theory in a rough and ready fashion in our description of the model theory for first-order languages. In a global language, we will want to be able to do this in a principled way.

The full theory of sets that one might find in a text book is very powerful and complex. However, the structures for which we are designing our constraint languages are all relatively simple; indeed, they are all finite, although we shall not need to assume this. Furthermore, we won't be needing much more than the simplest set of theoretic operations and constructions to express what we will want to express. Thus, all we need is enough set theory to meet these limited needs. We will provide this theory, along with some motivation and explication of the relevant concepts, in the next section.

Membership

A set, intuitively, is just a collection of things which themselves may or may not be sets. Usually we choose a set with the help of some predicate (e.g., the set of all prime numbers, all American citizens, or all track and field events in the 1988 Olympics). However, this is just for our benefit; any bunch of things, even if they can't be picked out by a common property, indeed even if they can't be picked out by us in any way at all (as is the case with, for example, most infinite collections of natural numbers), still form a set. We'll see shortly that we have to be a little more careful than this about the sets that we claim exist, but this at least gets our intuitions going about what sorts of things sets are.

The most basic relation a thing can bear to a set is that it can be a *member,* or *element*, of the set. Thus, the number 17 is a member of the set of all primes, George Bush is a member of the set of American citizens, and the (now unofficial) race in which Ben Johnson beat Carl Lewis is a member of the set of track and field events that took place in the 1988 Olympics. This special relation is nearly always represented by the symbol $\in$, and as with all the two place set-theoretic relations we will introduce, we will use infix rather than prefix notation. Thus, we will write $a \in b$ rather than $\in ab$.

Logically, sets are just individuals like any others; therefore, we will use constants to represent them. Furthermore, because not everything is a set, we will introduce a special predicate *SET* to abbreviate "is a set." Since it will often be convenient to say something general only about sets, we will set aside the letters r, s, and t (again, perhaps with subscripts and primes) to serve as special *set variables* that take only sets as values (and, as before, corresponding *sans serif* characters to serve as metavariables). In this manner, we will be able to say things about all and only sets without having to use the predicate "*SET*" explicitly. For example, suppose we want to say that the object a is a member of some set. Without these special set variables, we would have to express this as $\exists x(SET(x) \wedge a \in x)$; with them, we can simply write this as $\exists s(a \in s)$. Similarly, if we want to express that every set is

a member of some other set, without the set variables we have to write $\forall x(SET(x) \supset \exists y(SET(y) \land x \in y))$, whereas with them we can simply write $\forall r \exists s(r \in s)$. In general, and more abstractly, if s is any set variable that doesn't occur in a formula φ, $\forall x(SET(x) \supset \varphi)$ is equivalent to $\forall s \varphi_s^x$ and $\exists x(SET(x) \land \varphi)$ is equivalent to $\exists s \varphi_s^x$.

Frequently, we may want to say something φ about some or all members of a given set s. In our current grammar, this would be expressed as $\exists x(x \in s \land \varphi)$ or $\forall x(x \in s \supset \varphi)$, respectively. For convenience, we allow that these forms can be abbreviated as $(\exists x \in s)\varphi$ and $(\forall x \in s)\varphi$, respectively.

Basic Set Theoretic Axioms

Russell's Paradox. Sets combine and interact in many interesting ways; however, for deep and historically significant reasons, they do not react in every way one might expect. Consequently, we need to establish clear principles that tell precisely when such combinations and interactions can occur and what sets exist within a given domain. That is, we need some set theoretic axioms.

For example, consider the following famous paradox, known as *Russell's paradox* after the famous philosopher/logician Bertrand Russell who discovered it. As noted above, we often choose sets in ordinary contexts by means of some predicate or (more generally) description that holds of all and only the members of the set. Thus, for example, one might want to consider the set of all Texans over thirty-five who drink beer by means of the description "Texan over thirty-five who drinks beer," or more formally, the description $TEXAN(x) \land age\_of(x) > 35 \land DRINKS\_BEER(x)$. Let's use the notation $\{x \mid TEXAN(x) \land age\_of(x) > 35 \land DRINKS\_BEER(x)\}$ to name this set, and in general the notation $\{x \mid \varphi\}$ to name the set of things that satisfy the description φ. Now, intuitively, one would think that any such description φ with a single unbound variable picks out a corresponding set comprising the things that fit the description. After all, a set is just any bunch of things; so in particular the bunch of things satisfying a certain description is a set. Russell found that, intuitions to the contrary, this is not always so. Consider the description "set that doesn't have itself as a member"; that is, $s \notin s$ (remember that s is a set variable.) Intuitively, all sorts of sets satisfy this description: the set of horses isn't a horse and hence isn't a member of itself, the set of solar planets is not a planet, and so on. By the intuitive principle above, there is a set of all sets that satisfies this description; that is, there is the set $r = \{s \mid s \notin s\}$. But, is r a member of itself or isn't it? If it is, then, since r is the set of all sets that are not members of themselves, it follows that it is *not* a member of itself after all. If, on the other hand, it is not a member of itself, it satisfies the condition for membership in r (i.e., it actually *is* a member of itself). Either way, we contradict ourselves. So there can't be such a set as r after all, despite what our intuitions tell us.

The Axioms. The lesson here is that not just any collection of things we might have thought of is a set. Hence axioms are needed that avoid getting us into the same sort of logical trouble. For our purposes, we need surprisingly few: four axioms and one axiom schema. The first axiom, *extensionality*, tells us when two apparent sets are in fact identical, viz., when they have exactly the same members:

ST1 $\forall r \forall s (\forall x\,(x \in r \equiv x \in s) \supset r \approx s).$

That is, for all sets r and s, if for any object x, x is a member of r if and only if it is a member of s, then r and s are the same set.

The second axiom, *pairing*, states that any two objects (within a given domain) form a set:

ST2 $\forall x \forall y \exists s (s \approx \{x,y\}),$

where "$\{x,y\}$" is a name for the set that contains exactly the objects denoted by x and y (by extensionality there can be only one such set). Thus, to make this proper, we need to add to our vocabulary the left and right braces $\{,\}$, and to add to our grammar the rule that if $t_1 ,..., t_n$ are any terms, the expression $\{t_1,...,t_n\}$ is a term as well.[60]

The next axiom declares that the *union* of any set r exists; that is, the set whose elements are exactly the members of the members of r:

ST3 $\forall r \exists s "y (y \in s \equiv \exists t\,(t \in r \wedge y \in t)).$

In English, this axiom states that for any set r there exists a set s such that for any object y, y is a member of s if and only if there is a set t such that t is a member of r and the object y is a member of t. For a given set r, we will let $\cup r$ stand for the union of r ($\cup$ is thus a distinguished two-place function symbol, denoting the (partial) function that takes any set to its union). We will usually write $r \cup s$ for $\cup\{r,s\}$.

When one set a is a subset of another b (i.e., when all the members of a are members of b), we express this with a distinguished predicate as $a \subseteq b$. The fourth axiom says that the set of all subsets of any given set exists:

ST4 $\forall r \exists s \forall x (x \in s \equiv x \subseteq r).$

That is, for any set r there is a set s such that for any object x, x is a member of s just in case x is a subset of r. If $a \subseteq b$ and $a \neq b$, we say that a is a *proper* subset of b, and we express this as $a \subset b$. For any given set a, the set of all its subsets is called the *power set* of a. The (partial) function that takes each set to its power set will be denoted by the distinguished function symbol *pow*; thus, the power set of a will be denoted by *pow(a)*.

---

[60] Strictly speaking, we can think of ourselves as adding infinitely many new function symbols $f_1, f_2, ...$ to our language, where each f_n is an n-place function symbol, each of which can by convention be rewritten using the brace notation. The rewritten form of each f_n is thus evident by the fact that there are n terms between the braces (e.g., $\{a,b,c\}$ is the rewritten form of $f_3 abc$).

Finally, we come to our one set theoretic axiom *schema*, so-called because it actually stands for infinitely many axioms of the same general form, one for each formula of our language. It is called the axiom schema of *separation*, or *subsets*. The idea is quite simple: given a certain set *a* and some description φ in our language, we can separate out the set of all the members of *a* that satisfy the description. Formally, for any formula φ,

$ST5_\varphi \qquad \forall r \exists s'' x \in r (x \in s \equiv \varphi(x)),$

where $\varphi(x)$ is the result of replacing any given unbound variable in φ with x.[61]

Russell's Paradox Revisited. Given the separation axiom schema, we are able to reintroduce in a restricted form the notation for sets used in the brief discussion of Russell's paradox above. The paradox arises when one assumes sets can be generated *ex nihilo* with any given formula. Separation only allows one to use arbitrary formulas to form sets from the members of *previously given* sets and this eliminates the problem; in this light, in Russell's argument, for any given set *a* already proved to exist, one is allowed to assume only the existence of the set $\{s \mid s \in a \land s \notin s\}$; this assumption causes no problems at all. Consequently, we can safely add the following grammatical rule:[62] if φ is any formula, t any term, and x any variable, $\{x \mid x \in t \land \varphi\}$ is a term as well. Similar to what we allowed with certain types of quantified formulas, such terms can also be written as $\{x \in t \mid \varphi\}$.

Finitude and the Set of Natural Numbers

As noted, we are assuming the existence of the natural numbers. It will prove very useful, then, to assume in addition that they jointly form a set; this is not provable from the above axioms. The easiest way to assure this is to add an axiom that declares this explicitly:

$NN \qquad \exists s \forall x (x \in s \equiv NUM(x)).$

---

[61] Assuming of course *x* does not become bound in the process. If it does, we can always replace it in the above schema with a new variable not occurring in j.

[62] Or, more cautiously, it appears that we can do so safely for all we can tell. Due to Godel's famous *second incompleteness theorem*, there is no way to *prove* there aren't other hitherto undiscovered paradoxes lurking in the theory of sets; that is, we cannot prove its consistency (at least, not without begging the question by proving it in a theory that is at least as dubious). However, the great success of the theory over the past eighty-five years and the absence of any new paradoxes despite extensive use and scrutiny of the theory has given logicians great confidence that it is in fact consistent, even if we shall never know this with utter certainty.

That is, there exists a set s such that for any object x, x is an element of s if and only if x is a natural number. By the axiom of extensionality, there can be only one such set; we shall call this set "N."

We are now able to define another useful notion. As noted, the structures we are interested in examining are all finite. Nonetheless, it will still be important to be able to *say* explicitly that they are finite; hence, we need to be able to express the concept of finitude. We can do this with the help of the set N. Specifically,

Def 5: $FINITE(s) =_{df} \exists n \in N \ (s \sim \{m \in N\} \mid m < n\})$,

where $t \sim r$ means intuitively that t and r are the same size; that is, there is a one-to-one correspondence between them. (This latter notion can also be defined straightforwardly with the set theoretic apparatus at our disposal.) Thus, a set is finite just in case it is the same size as the set that contains all and only the natural numbers less than a given natural number n. The number n is said to be the *cardinality* of the set.

Difference, Intersection, and the Empty Set

Many interesting and important facts about sets are derivable from the above axioms. One is that the existence of the *difference a-b* of two sets a and b (i.e., the set of elements of a that are not in b) is a new two-place functions symbol. It is easy to prove that a-b exists because, by separation, there is an s that contains just those elements of a that are not in b (i.e., an s such that for all $x \in a$, $x \in s$ _ $x \notin b$).

Another useful theorem is that the *intersection* of any two sets exists, where the intersection of sets a and b is just the set of all objects that a and b both have as members. We'll refer to this set as $a \cap b$, making use of the distinguished two-place function symbol $\cap$. The proof that $a \cap b$ exists is also easy: by union, $\cup\{a,b\}$ exists; by separation, we pull out the set of all $x \in \cup\{a,b\}$ such that both $x \in a$ and $x \in b$. In general, we can show that the intersection of any number of sets exists in essentially the same way.

Notice that often there might be no elements common to two sets. Nonetheless, their intersection is a perfectly good set: the empty set. We can prove the existence of the empty set a bit more formally, as follows. We know there are sets because first-order logic guarantees the existence of at least one object a, and by pairing it follows that the singleton set $\{a\}$ exists. By the schema of separation, letting φ be the formula $\neg(x \approx x))$, there is a set s that contains all the members x of $\{a\}$ such that $x \neq x$ (i.e., all the members of $\{a\}$ that are not identical to themselves). Of course there are no members of $\{a\}$ that fit that description; therefore, s is a set with no members — the empty set. Following the usual practice, we will use the constant "$\varnothing$" to refer to this set. Two sets r and s are said to be *disjoint* if they have no members in common (i.e., if $r \cup s = \varnothing$). A set s of sets is said to be *pairwise disjoint* if any two members of s are disjoint.

Functions and Ordered n-tuples

Set theory enables us to provide an elegant account of certain other important notions. First, an extremely versatile and useful notion is that of an *ordered pair*. An ordered pair is like a set of two elements, except that unlike a set, which is an unordered collection, there is a first member and a second member. Thus, where $\langle a,b \rangle$ stands for the ordered pair whose first element is a and whose second element is b, the important characteristic about ordered pairs is that they satisfy the following principle:

OP $\forall x \forall y \forall z \forall w (\langle x,y \rangle \approx \langle z,w \rangle \supset (x \approx z \land y \approx w))$.

That is, ordered pairs are identical only if their first elements are identical and their second elements are identical; thus, they are identical only if, like any set, they have the same elements, *and*, unlike sets — which have further structure beyond their elements, those elements occur in the same order. Therefore, the way we write down names for the members of an ordered pair, unlike sets, is significant because the first name we write down signifies the first element of the pair, and the second name the second element. For example, whereas $\{a,b\} \approx \{b,a\}$, we have in the case of ordered pairs that $\langle a,b \rangle \neq \langle b,a \rangle$.

As it happens, we needn't introduce ordered pairs as a new sort of object because, with a little set theory, it is easy to define them as sets of a certain sort. We will adopt the usual definition of what are called "Kurotowski" ordered pairs. Specifically, we define the ordered pair $\langle a,b \rangle$ to be the set $\{\{a\},\{a,b\}\}$. It is easy to check that ordered pairs so defined satisfy the above principle. Given the definition of an ordered pair, we can now define the notion of an ordered n-tuple $\langle a_1,...,a_n \rangle$ recursively to be the ordered pair $\langle \langle a_1,...,a_{n-1} \rangle,a_n \rangle$.

Given the notion of an ordered n-tuple, we can give a more precise account of the notion of a function. A one-place function f from one set r to another s is just a mapping that takes each element a of r (or some subset of r, if f is partial) to an element $b = f(x)$ of s. Thus, we can think of such a function as simply a set of ordered pairs $\langle a,b \rangle$ where, intuitively, b is the element that a is mapped to by the function f. More generally, an n-place function is a set of ordered $n+1$-tuples $\langle a_1,..., a_n, a_{n+1} \rangle$ where, intuitively again, a_{n+1} is the object that $a_1,..., a_n$ are mapped to by the function. Thus, functions are simply a type of set. The set of all one-place functions from one set r to another s will be denoted by $^r s$.

Number Theory

We will assume that a global language will contain the basic resources of arithmetic: a distinguished predicate *NUM*; the numerals; the usual function symbols $+$, $\cdot$, and *exp*; and enough axiomatic power to prove basic arithmetical facts. Thus, the intended semantics for any global language will always contain the natural numbers, with these syntactic items receiving the obvious interpretations.

317

The Intended Semantics: The Cumulative Hierarchy of Sets

The above gives a good idea of how sets combine and interact, and what sets we can suppose there to be. However, it doesn't really provide much of a panorama — much of an idea of the intended semantics — for set theory and hence for constraint languages generally. The intended picture of the structure of sets within a given domain is known as the *iterative*, or *cumulative*, conception of set. On this conception, sets are hierarchical; they come in *levels*. The lowest level L_0 (set theorists like to start counting with 0) consists of our initial set of *atoms* or *urelements* (i.e., things that are not themselves sets, for example, numbers, people, machines, buildings, strings, database records, countries, etc.). The next level, L_1 consists of all possible subsets of L_0 together with the urelements, i.e., $L_1 = pow(L_0) \cup L_0$. The next level, L_2, consists of all possible subsets of L_1 together with all the elements L_1. In general, $L_{n+1} = pow(L_n) \cup L_n$. Each level is thus cumulative; that is, each level pulls up the elements of the previous level to join all the sets that could be formed out of those elements. And so it continues through the sequence of natural numbers. The intended semantics for a given constraint language, sets and all, is just the union of all these levels; that is, $\cup_{i \in N} L_i$.[63]

---

[63] Though this is not anything we can say in the formal constraint language itself because we can only use it to talk about things *within* its semantic domain — failure to realize this ever-present semantic limitation is in fact what lies behind Russell's paradox.

APPENDIX E
SITUATION THEORY

Situations

In situation theory, situations are (typically) concrete, spatially and temporally extended pieces of the real world. Thus, we take a situation s to be characterized by the *interval of (space) time* **int(s)** over which it occurs, a certain set of objects **dom(s)** called its *domain* (intuitively, the set of objects that exist throughout **int(s)** in s), and an *extension function* **ext$_s$** that describes the "qualitative character" of s (i.e., the properties exhibited by the objects in s and the relations in which they stand).

This threefold characterization of situations provides a good deal of flexibility in defining a number of auxiliary notions that are important in process modeling. For instance, the information in a given domain is often dependent on the *perspective* of a given agent; a shop floor manager is likely to have a different perspective on a given manufacturing process than is the CEO of the company. We can capture differing perspectives by means of different but related situations in any of several ways.[64] For instance, a rather strong notion of differing perspectives can be captured via two (or more) situations s and s' that share the same domain and the same spatio-temporal interval but differ in regard to the extensions of certain (intuitively, non-perspectivally-invariant) properties and relations. Obvious examples are situations that involve different implicit frames of reference. A more subtle example — and one more likely in the context of business process modeling — involves differences of perspective that arise not from different spatial orientations but rather from different *conceptual* orientations. Consider, for instance, a given run of a manufacturing process viewed from the perspective of a shop-floor manager as opposed to that of a financial officer. Though they are perceiving the same objects in the same interval and may agree on some of the properties and relations of those objects (e.g., conceptual invariants like color in normal light, machine-type, etc.), the shop-floor manager will ascribe properties and relations to objects in the situation of which the financial officer is not even cognizant (e.g., *Rockwell-hardness*) and vice-versa (e.g., *depreciation rate*). The idea of properties and relations that have no purchase in a given situation (like *depreciation rate* in the shop-floor manager's perspective) is captured in our account by allowing properties and relations to be *undefined on* (as opposed simply to being *false of*) certain objects in the situation.

---

[64] Alternatively, we could define situations more broadly to be *everything* occurring in a given spatio-temporal interval and define what we're calling situations to be perspectives. Obviously, this is nothing more than a terminological issue.

The *granularity* of a situation is an important notion related to the idea of perspective. Intuitively, the same spatio-temporal interval can contain numerous situations that differ only with regard to their granularity — that is, with regard to the density of the mesh by means of which objects in that interval are distinguished from one another. One version of granularity can be captured using our apparatus together with a distinguished *part-of* relation that is defined on the domain of objects. In this version, one moves from a situation s to a situation s′ of finer granularity by including in s′ parts of objects in s that were ignored in that situation without ignoring any of the original objects in s. More exactly, we can say that one situation s∉ is finer-grained than another s if and only if (1) they share the same interval (i.e., $int(s') = int(s)$), (2) $dom(s) \subseteq dom(s∉)$, (3) any object in $dom(s')$ but not in $dom(s)$ is part of some object in $dom(s)$, and (4) the extensions of properties and relations with respect to members of s remain the same except for those that are "sensitive" to shifts in granularity (e.g., the property of having parts).[65]

A related notion that can be represented in this account of situations is that of a *decomposition* of one situation into finer-grained sub-situations. Again, there are several related ways of capturing this idea using our apparatus. One rather strict way, for the purpose of illustrating the approach, is as follows. A sequence $\langle s_1,...,s_n \rangle$ of situations is a decomposition of a given situation s only if (1) the end of $int(s_i)$ is the beginning of $int(s_{i+1})$ for $i < n$, (2) $int(s) = \bigcup_{i \leq n} int(s_i)$, (3) $dom(s) \subseteq \bigcup_{i \leq n} dom(s_i)$, and (4) the extensions of properties and relations with respect to members of s∉ remain the same except for those that are "sensitive" to decomposition. That is, $\langle s_1,...,s_n \rangle$ is a decomposition of s only if each s_i (except the last) is immediately followed by its successor in the sequence, the s_i jointly occur over precisely the same interval as s, the objects that occur in the s_i jointly compose at least the objects that occur in $dom(s)$, and properties and relations remain constant in the manner indicated.

Infons

Basic Infons

As noted, a situation is individuated by the pieces of information — the *infons* — that it supports or that hold within it. The *basic* infons in a given situation s consist of all possible legitimate units of information of the form:

objects $a_1, ..., a_n$ stand in relation r,

and

---

[65] For the most part, properties of this sort will just have to be tagged as such ahead of time because there is no straightforward way of cashing this notion formally.

objects $a_1, ..., a_n$ do not stand in relation r,

where r and the a_i are all constituents of s. We represent these infons as the $n+2$-tuples $\langle\langle r, a_1, ..., a_n, +\rangle\rangle$ and $\langle\langle r, a_1,...,a_n,-\rangle\rangle$, respectively. A basic "positive" infon $\langle\langle r, a_1, ..., a_n, +\rangle\rangle$ holds in s just in case its component objects $a_1, ..., a_n$ stand in the relation r in s, and a basic "negative" infon $\langle\langle r, a_1,...,a_n,-\rangle\rangle$ holds in s just in case $a_1, ..., a_n$ are present in s, are appropriate for r in s, but do not stand in that relation in s.

Complex Infons and the Infon Algebra

Boolean Infons

Complex infons are generated from basic infons by means of a number of boolean and quantificational operations. Specifically, to begin with, for any two infons i and i', there are also the "boolean" infons $i \otimes i'$ and $i \oplus i'$: their *conjunction* (roughly, the information that both i and i' hold) and their *disjunction* (roughly, the information that either i or i' holds). We impose a number of constraints on the behavior of Boolean operators f and Å to guarantee that our formal models of infons reflect the desired characteristics of these entities. For instance, both operations are *idempotent, commutative,* and *associative* (i.e., letting * stand for either operation) for any infons i, i', i^2:

1. $i * i = i$

2. $i * i' = i' * i$

3. $(i * i') * i^2 = i * (i' * i^2)$.

The motivation for all three conditions is that, intuitively, certain distinct ways of conjoining and disjoining infons either yield no new information (Condition 1) or yield information that can be obtained by slightly different means (Conditions 2 and 3).

The next three conditions are intended to capture jointly the fine-grained internal structure of infons. The guiding intuition here is that each algebraic operation "builds up" more complex infons from less complex infons in a way unique to that operation and in such a way as to preserve the structure of the original infons. The uniqueness of each operation's action is captured in the condition that no conjunctive infon — i.e., no infon i such that $i = j \otimes j'$, for distinct j, j' — is disjunctive, and vice versa. More precisely,

4.   If $i \neq i', j \neq j'$, then $i \otimes i' \neq j \oplus j'$.

Two further conditions ensure that structure is preserved. Because basic infons are, intuitively, the atoms from which complex infons are constructed, the first of the following conditions asserts that a basic infon is not the

conjunction or disjunction of any two distinct infons. (Compare the condition in elementary number theory that zero is not the successor of any number.) More precisely,

5. If i is a basic infon, then $i' * i^2 = i$ only if $i = i'$

(= i^2 as well, by commutativity). The second condition captures a similar idea with regard to compound infons, viz., that, roughly speaking, conjunctions and disjunctions of distinct infons always yield infons that "inherit" the structure of both the original infons; no structure should be lost via *. In particular, we want to prevent * from "looping back" on a given pair of infons to any less complex or structurally unrelated infon; rather, * should always "generate" new, more complex infons that inherit the structure of the less complex infons from which they are built up. (Thought of in this way, the condition can be likened to the one-to-one condition on the successor function in elementary number theory which ensures that $S(n)$ never loops back to a smaller number but always "generates" new, larger numbers from previously given, smaller numbers.) To get at the idea, define an infon i to be *-*basic* with respect to an infon i' just in case there is an infon j such that $i * j = i'$ and for any infons j, j', $j * j' = i$ only if i = j (= j', by commutativity). So the ⊗-basic "elements" of a conjunctive infon i are those infons which are "conjuncts" of i but are not themselves conjunctions (except, trivially, by idempotence, of themselves); similarly for disjunctive infons. Given this, we stipulate that

6. $i * i' = j * j'$ if and only if any infon that is *-basic with respect to either i or i' is *-basic with respect to either j or j', and vice versa.

Intuitively, the only way to generate a conjunctive infon in two different ways is by building it up from the same set of nonconjunctive components; similarly for disjunctions. Thus, structure is always preserved by the binary operations. To illustrate, suppose i is a disjunctive infon $i' ⊕ i^2$ (i',i²_ i) and j and k are distinct basic infons (so k is not ⊗-basic with respect to either i or j), and let us assume that $(i ⊗ j) ⊗ k = i ⊗ j$ (i.e., we "lose" k when we conjoin it to $i ⊗ j$, and ⊗ "loops back" to $i ⊗ j$ when it takes $i ⊗ j$ and k as arguments). Using Condition 5, it is easy to show that k, being atomic, is ⊗-basic with respect to itself. Hence, by Condition 6, it must be ⊗-basic with respect to either i or j. Thus, by definition, there is some k' such that $k ⊗ k' = i$ or $k ⊗ k' = j$. However, neither is possible because i is a disjunctive infon and, by Condition 4, $k ⊗ k' ≠ i$; similarly, since by assumption j is basic and $j ≠ k$, by Condition 5 we have that $k ⊗ k' ≠ j$. Therefore, $(i ⊗ j) ⊗ k ≠ i ⊗ j$.

Having characterized how the Boolean operations affect the internal structure of infons, we now want to characterize how they affect their "standing" in a given situation with respect to the *supports* relation. Letting T, ⊥, and ↑ indicate that an infon holds, fails, or is undefined, respectively, in a given situation s, we can characterize how a Boolean infon i varies with respect to these possible values at s as a function of the values of j and j' in s by the table in Figure E-1 (since both operations are commutative, we omit redundant cases):

j	j'	j ⊗ j'	j ⊕ j'
T	T	T	T
T	⊥	⊥	T
T	↑	↑	↑
⊥	⊥	⊥	⊥
⊥	↑	↑	↑
↑	↑	↑	↑

Figure E-1

Three-Valued Truth Table for Boolean Infons

Indeterminates and Quantified Infons

As noted, in addition to the Boolean operations there are also operations corresponding to existential and universal quantification. Intuitively, quantifiers bind "argument roles" in properties and relations. In situation theory, these argument roles can be represented explicitly in infons by means of *indeterminates*, nondescript "variable" objects of a certain kind that serve as place-holders for genuine objects of that kind. Infons containing indeterminates are known as *parametric* infons. When the indeterminate placeholders in an indeterminate infon are replaced by objects of the appropriate type, a parametric infon results. To illustrate, let **r** be the property of being an engineer, and let **x** be an indeterminate object; then $\langle\langle r,x,+\rangle\rangle$ is the indeterminate piece of information that **x** is an engineer. Suppose now we replace **x** with an appropriate argument for **r** (e.g., Malcolm Baldridge); then, the nonparametric infon $\langle\langle r, \text{Malcolm}, +\rangle\rangle$ results. Another way of yielding a nonparametric infon from an parametric infon, rather than by replacing an indeterminate, is to bind the indeterminate with a quantifier. Thus, if we bind the argument role **x** in $\langle\langle r,x,+\rangle\rangle$ with the existential quantifier **Some x**, an existentially quantified infon results — the infon that some object **x** is such that **x** is an engineer; formally, $\Sigma_x\langle\langle r,x,+\rangle\rangle$, where '$\Sigma_x$' represents the quantifier **Some x**. Similar remarks apply to the universal quantification operators Π_x. Note that indeterminates are legitimate situation theoretic objects; hence for each indeterminate **x** there are distinct universal and existential quantifiers Σ_x and Π_x.

As with the Boolean operators, it is important to characterize how quantification operators affect the structure of the infons they operate on in a way that accords with both intuition and representational needs. We begin with several characteristics relating to the identity and individuation of quantified infons.

323

7.   For any infon j, Q_x is defined on j if and only if x is an (unbound) indeterminate of j.

Thus, referring to the earlier example above, the quantifier Q_y is not defined on $\langle\langle r,x,+\rangle\rangle$ (assuming y is a distinct indeterminate from x). The next condition builds on this and asserts that the application of a quantification operator on an infon always yields a new infon. Formally,

8.   For any infon j, if Q_x is defined on j, then $j \neq Q_x j$.

The next condition essentially stipulates that the order of application of any finite number of distinct quantification operators of the same type (i.e., either all of the form Σ_x or all of the form Π_x) is irrelevant; any way of applying them sequentially should yield precisely the same piece of information. Formally put, let $Q_{x_1}...Q_{x_n}$ be a sequence of pairwise distinct quantification operators of the same type, and let $Q_{y_1}...Q_{y_n}$ be any permutation of $Q_{x_1}...Q_{x_n}$; then

9. $Q_{x_1}...Q_{x_n} j = Q_{y_1}...Q_{y_n} j$.

The next condition ensures that quantified infons are sufficiently fine-grained; specifically (and informally),

10.   No universally quantified infon is identical to any existentially quantified infon, and no quantified infon is identical to any basic infon or any "Boolean" infon.

To characterize when a quantified infon is supported by a given situation it is necessary to introduce an auxiliary notion, that of an *anchor*. A function f that maps the indeterminates of a parametric infon i to objects of the appropriate type is called an *anchor* for i; the infon that results from replacing each indeterminate x in i by $f(x)$ is indicated by $i[f]$. Given this, where $\Pi_x j$ is an infon with no indeterminates, $\Pi_x j$ holds in a situation s just in case, for *every* anchor f for j taking its values in $\mathbf{dom}(s)$, $j[f]$ holds. $\Pi_x j$ fails in s if it is a legitimate piece of information in s[66] and $j[f]$ does not hold in s for *some* anchor f for j that takes its values in $\mathbf{dom}(s)$. Similar definitions with appropriate changes (e.g., "some" for "every") apply to existentially quantified infons $\Pi_x j$.

Structural Similarity

Infons that are structurally identical except for their "choice" of indeterminates are said to be *similar* (with respect to those indeterminates). However, the apparatus to make this precise, though not conceptually difficult, is

---

[66] The notion of being a legitimate piece of information in an situation s is defined rigorously in our process information formalization.

quite elaborate. For the purposes of this report, we shall just discuss two further conditions on the identity of infons very generally. First, since similar infons differ only with respect to a choice of indeterminates, it is intuitive that:

11.   The information that results from anchoring corresponding indeterminates in similar infons is identical; that is, where f and f' are two such anchors for similar infons i and i', $i[f] = i[f']$.

The second principle is concerned primarily with the effects of "binding" an indeterminate by means of a quantification operator. Indeterminates in situation theory are distinct individuals (of a certain unusual sort); hence it is intuitive, as well as theoretically important, to distinguish structurally similar infons containing different indeterminates. For instance, in the otherwise identical parametric infons $\langle\langle \text{running}, x, t, + \rangle\rangle$ and $\langle\langle \text{running}, y, t, + \rangle\rangle$, the indeterminates x and y might need to be anchored to different persons in the same situation to model, for example, the semantics of a sentence uttered about a race occurring in the situation. However, one effect of a quantification operator is to nullify the distinctiveness of the quantified indeterminate in an infon. Thus, when the indeterminates x and y are quantified as in, for example, the infons $\Sigma_x \langle\langle \text{running}, x, t, + \rangle\rangle$ and $\Sigma_y \langle\langle \text{running}, y, t, + \rangle\rangle$, the bound parameters play precisely the same roles; both infons hold in a given situation (given an anchor for t) under precisely the same conditions, viz., if something among the elements of u is running at t. Thus, there is no reason not to think of them as precisely the same piece of information.[67] More generally, the second principle is that

12.   Infons that differ only in their quantified indeterminates are identical.

As with informal conditions above, Condition 12 can be made formally precise.

Situation Types and Object Types

As noted in previously in this paper, type-level information and the correspondence between tokens and their type are represented elegantly in situation theory in the form of object types and situation types. A situation type T is specified in terms of an indeterminate situation s and a set Γ of infons and is indicated by the notation '$[s \mid \Gamma]$', read "the type of situation s such that s supports (every element of) Γ." A situation s is *of type* T just in case there is an anchor f such that $f(x) = s$ and $i[f]$ holds in s for each element $i \in \Gamma$. In such case, we also say s is of type T *under* f.

Analogously, an object type T is specified in terms of an indeterminate object x and a set Γ of infons, and is indicated by the notation '$[x \mid \Gamma]$,' read "the type of object x such that (every element of) Γ holds." An object a is *of type* $T = [x \mid \Gamma]$ in a situation s just in case there is an anchor f such that $f(x) = a$ and $i[f]$ holds in s, for each

---

[67] Compare the notion of alphabetic variance in first-order logic.

element i∈G. In such case, we also say a is of type **T** *under* **f**. For example, let **G** consist of two infons: the quantified infon $S_y \langle\langle \text{married-to},x,y,+ \rangle\rangle$ and the basic infon $\langle\langle \text{happy},x,+ \rangle\rangle$. Now let **T** be the type [**x** | **G**], that is, the type:

[**x** | {$S_y \langle\langle \text{married-to},x,y,+ \rangle\rangle$, $\langle\langle \text{man},x,+ \rangle\rangle$}].

Intuitively, this is the type of all married men; cashing the definition above, we have that an object **a** will be of type **T** in a given situation **s** (in whose domain, of course, **a** exists) just in case there is an anchor **f** such that **f**(**x**) = **a** and $\Sigma_y \langle\langle \text{married-to},x,y,+ \rangle\rangle$[**f**] and $\langle\langle \text{man},x,+ \rangle\rangle$[**f**] hold in **s**, that is, just in case $\Sigma_y \langle\langle \text{married-to},a,y,+ \rangle\rangle$ and $\langle\langle \text{man},a,+ \rangle\rangle$ both hold in **s** (i.e., just in case **a** is a married man in **s**). Under these conditions, the situation **s** is also of type **T'** = [**s'** | Γ] because there is an anchor, viz., **f**, such that the elements of Γ hold in **s** relative to **f**.

Constraints and Process Descriptions

In situation theory, a constraint is a certain kind of relation between types of situations: the type of situation in which the antecedent conditions k_1, ..., k_n in question holds and the type of situation in which C consequently holds. Expressing the conditions k_1, ..., k_n and k as sets Γ, Γ' of (typically, parametric) infons, respectively, we express such constraints explicitly as:

[**s** | Γ] ⇒ [**s'** | Γ'].

A constraint is said to be *satisfied* in a given system S just in case, for any situation **s** of type **T** = [**s** | Γ] under **f**, there is a situation **s'** and an anchor **f'** for **T'** = [**s'** | Γ'] such that (1) **f'** extends **f**,[68] and (2) **s'** is of type **T'** under **f'**.[69] The requirement that **f'** agree with **f** on the indeterminates in **T** is included to ensure that anchorings do not change from the antecedent type to the consequent type with respect to the indeterminates that occur in both types.

This requirement is important because it provides a mechanism that enables one to impose constraints on objects across situations. For instance, it is a logical constraint that any bachelor is a man. In situation theoretic terms:

[**s** | $\langle\langle \text{bachelor},x,+ \rangle\rangle$] ⇒ [**s** | $\langle\langle \text{man},x,+ \rangle\rangle$].

---

[68] That is, **f'** agrees with **f** on every value on which **f** is defined.

[69] Most accounts of situation theory require that **f'** be an *extension* of **f**, but we have found this condition to be too strong because **f** might already be defined on indeterminates that don't occur in **T** but do occur in **T'**, possibly leading to undesirable anchorings of those indeterminates.

Because both indeterminates s and x occur in both types, the condition for constraint satisfaction means that, if this constraint holds, then if x is anchored to a bachelor b in a given situation to which s is anchored, then (since s is also the situation indeterminate in the consequent type) that very same situation must be one in which that very same object b is a man.

However, as we will see below, there are cases where we don't want a condition this strong. To accommodate such cases, we introduce a weaker constraining relation $\Rightarrow^{\alpha}_{\Delta}$, where a is a set of indeterminates and D a set of infons, and we define $T \Rightarrow^{\alpha}_{\Delta} T'$ to hold in a system just in case, for any situation s_1 of type $T = [s \mid \Gamma]$ under f, there is a situation s_2 and an anchor f' for $T' = [s' \mid \Gamma']$ such that (1) f' extends f *except* perhaps with respect to the indeterminates in a, and (2) s_2 is of type $T'_{\Delta} = [s' \mid \Gamma' \cup D]$ under f'. (D thus enables one to impose additional conditions on instances of T and T' for the constraining relation to hold.) The original relation $\Rightarrow$ is now simply the special case of $\Rightarrow^{\alpha}_{\Delta}$ where a = D = $\varnothing$.

Now, armed with the notion of a constraint, it is straightforward to give a precise definition of a process (i.e., a process type): *a process is a set of systematically related constraints*. Thus, a process model is a representation of a process so understood. The types that constitute the constraints in a process, of course, represent the various activities (at some level of granularity) that constitute the process, and the relations between them are the temporal orderings and other salient relations that determine the structure of the process. A process model will typically contain some representational entity E_T to stand for each type T in the process. Semantically put, then, T is the *meaning* of the representation E_T. Arrows (as in IDEF3) or some other representational device are generally used explicitly to indicate temporal relations between these types. Richer modeling methods provide means of increasing the expressive capacities of the basic representations of the method, thereby enabling a modeler to add more detail to his or her process descriptions. This is the purpose of the IDEF3 elaboration language.

For example, consider a simple process involving a paint/dry loop in which a widget is painted, dried, then tested. If the paint job is adequate, the widget moves down the production line; otherwise it is shunted back into the process. We assume different widgets can be undergoing different activities in the process concurrently. This process can be represented in a very coarse manner using a simple graphical diagram as shown in Figure E-2.
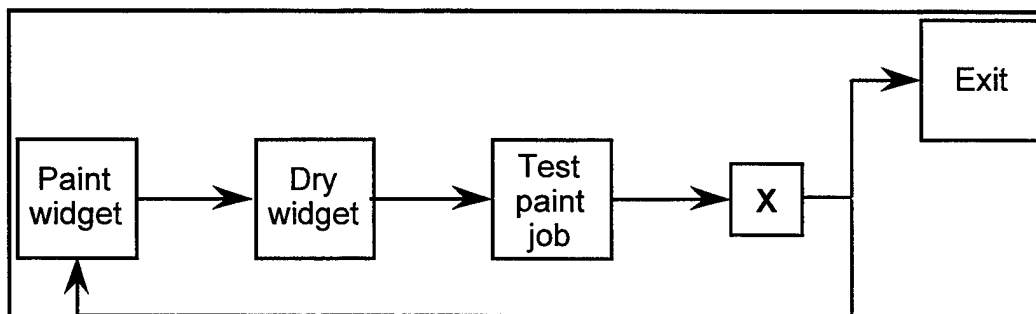

Figure E-2

Paint/Dry Cycle

The labels in the larger boxes provide a semantics for the diagram by virtue of the conventional English meaning of the words; however, this is semantics only in the loosest sense. Formally, given our analysis of process models, all we know is that each label is a name of a certain situation type whose details are only implicit in the label; let us make them explicit. For convenience, let us refer to the types referenced by the labels in the larger boxes, from left to right, as T_1, T_2, T_3, and T_4. Making their content explicit is a matter of describing their internal structure (i.e., describing the objects in the domains of situations of those types and how those objects are related within those situations). Suppose that in T_1 in our small example, a specific paint sprayer p paints each widget w that comes down the production line. That is, in situations s of type T_1, p and a widget w stand in the *paints* relation. Thus:

$$T_1 = [s_1 \mid \{\langle\langle \textit{paint-sprayer}, p, + \rangle\rangle, \langle\langle \textit{widget}, w, + \rangle\rangle, \langle\langle \textit{paints}, p, w, + \rangle\rangle\}],$$

where s_1 is a situation indeterminate, and w an object indeterminate. Situations of type T_2 contain a widget being dried by a dryer d. This in itself isn't enough. We also need to indicate that, in the context of the overarching process, each instance s_2 of T_2 follows (immediately, let us for simplicity suppose) an instance s_1 of T_1, and that the widget in s_2 is the same widget as the one in s_1. The first of these needs can be met in a variety of ways. Perhaps the easiest is to include all temporal intervals in the domain of every situation s, not just those contained in $\mathbf{int}(s)$. Then we can simply add the requisite temporal information to T_2. This is the easiest route for the purposes of this paper.[70] More precisely, we explicitly include in T_2 the information that the end of $\mathbf{int}(s_1)$ is identical to the beginning of $\mathbf{int}(s_2)$. Thus,

$$T_2 = [s_2 \mid \{\langle\langle \textit{dryer}, d, + \rangle\rangle \langle\langle \textit{dries}, d, w, + \rangle\rangle, \langle\langle \textit{equals}, \textit{end}(\textit{int}_1(s_1)), \textit{beg}(\textit{int})(s_2))\rangle\rangle\}].$$

To connect T_1 and T_2 properly as part of the overall process, we form the constraint $T_1 \Rightarrow T_2$. This constraint guarantees that if there is an instance s_1 of T_1, there must also be an instance s_2 of T_2. Furthermore, the temporal constraint in T_2 guarantees that s_2 must begin as soon as s_1 ends, and the definition of constraint satisfaction will guarantee that w must be anchored to the same widget in both processes, thus filling the second need noted above.

In the same manner, we define T_2. For the purposes of this paper, let's eliminate unnecessary detail and simply lump the testing procedure into a simple infon to the effect that a tester e tests the paint job on w. We also add a temporal constraint analogous to the one above. Thus,

---

[70] Intuitively, however, information about the temporal relations between situations is somehow at a higher logical level and, hence, should perhaps be captured differently. We are currently examining other options that do better justice to this intuition.

$T_3 = [s_3 \mid \{\langle\langle tester, \mathbf{e}, + \rangle\rangle, \langle\langle tests, \mathbf{e}, \mathbf{w}, + \rangle\rangle, \langle\langle equals, end(\mathbf{int}(s_2)), beg(\mathbf{int}(s_3))\rangle\rangle\}]$,

and we add the constraint $T_2 \Rightarrow T_3$. Finally, to capture the looping nature of the process, we add the disjunctive constraint $T_3 \Rightarrow T_4 \oplus T_3 \Rightarrow_\Delta^\alpha T_1$, where $a = \{s_1\}$, $D = \{\langle\langle equals, end(\mathbf{int}(s_3)), beg(\mathbf{int}(s_1))\rangle\rangle\}$, and $\oplus$ is exclusive disjunction. This constraint is satisfied in the system just in case exactly one of $T_3 \Rightarrow T_4$ and $T_3 \Rightarrow_\Delta^\alpha T_1$ is satisfied, thus capturing the intended logic of the disjunctive branch in the process model. The more general relation $\Rightarrow_\Delta^\alpha$ enables the constraint between T_3 and T_1 to be satisfied in a way that allows the situation indeterminate s_1 in T_1 to be "updated" when the entire system of types is anchored to an instance of the process that loops;[71] the added condition $\langle\langle equals, end(\mathbf{int}(s_3)), beg(\mathbf{int}(s_1))\rangle\rangle$ then ensures in such instances of the process that the proper temporal relationship holds between instances of T_3 and the new instances of T_1.

---

[71] Recall that the definition of $\Rightarrow$ requires that each anchor for the types in a chain of constraints must extend its predecessor.

This page intentionally left blank.

APPENDIX F
A FORMALIZATION OF MODEL TRANSLATION

Appendix F provides a formalization of model translation using classical semantic techniques. Though powerful, the limitations of model translation and the need for a broader situation theoretic approach have been documented in that section as well.

A global representational medium like KIF or the NIRS must be capable of expressing any possible piece of information expressible in any of a set of given target languages. For then, and only then, could one be guaranteed a translation from one knowledge base to another whenever one is possible. Theoretically, we can capture this idea by means of the notion of an *interpretation* of one first-order theory into another. Following Enderton (1972), an interpretation of a language L0 into a theory T1 (in a language L1) is a function π on the quantifiers, constants, function symbols, and predicates of L0 such that:

1. $\pi_\forall$ is a formula of L1 in which at most x occurs free, such that T1 entails $\exists x \pi_\forall$.

2.   For each *n*-place predicate P, p_π is a formula of L1 in which at most the variables $x_1,..., x_n$ occur free.

3.   For each *n*-place function symbol f, π_f is a formula of L1 in which at most $x_1,...,x_{n+1}$ occur free such that T1 entails $\pi_\forall(x_1)$ & ... & $\pi_\forall(x_n)$ ... $\exists x \pi_\forall$ & $\forall x_{n+1}(\pi_f(x_1,...,x_{n+1}) \equiv x_{n+1} = x)$.

Intuitively, the idea behind (1) is that the formula $\pi_\forall$ picks out a class of things to serve as the domain of individuals in a model of L0 relative to a given model M of T1, viz., the class $\text{Dom}_\pi = \{b \mid \pi_\forall$ is true in M when **b** is assigned to the variable x}. Accordingly, if $\pi_\forall$ is to do the job, T1 must prove that the class it picks out is nonempty (since the classical semantics for first-order logic doesn't allow empty domains).

The idea behind (2), of course, is that π_P is a formula that picks out the extension of P in a model of L0, viz., again relative to a model M of T1, $\{\langle a_1,..., a_n\rangle \in \text{Dom}_\pi{}^n \mid \pi_P$ is true in M when a_i is assigned to the variable $x_i\}$. similarly, the idea behind (3) is that p maps each function symbol f into a formula that picks out a "functional" relation on the domain Dom_π (i.e., a class of *n*+1-tuples out of $\text{Dom}_\pi{}^{n+1}$ whose first n elements represent the "input" to a function and whose $n+1^{th}$ element represents the "value" of the function on that input). A separate clause for constants can be avoided by taking them to be 0-place function symbols.

Let π be an interpretation of L0 into T1, let T0 be a theory in L0, and for a given sentence φ of L0, let t be a function that takes each formula φ of L1 to the formula t*(φ) of L1 that "corresponds" to φ in the obvious way under π. (This can be defined recursively in a fully rigorous fashion.) Then we say that t is an interpretation of the *theory* T0 in T1 just in case T0 ④ φ iff T1 ④ t(φ), for any sentence φ of L0.

Of course, the idea is that an interpretation (a recursive one, anyway) provides a procedure for taking any sentence in the language of one theory and returning another sentence in the language of another theory that in some sense "says the same thing." This can be seen as a special case of the general idea of a translator from a given knowledge base B, written in a given KR language L, into ISyCL. While L might not be a first-order language per se (e.g., the expressively rather weak graphical language of E/R database schema diagrams [Chen, 1976]), insofar as it can carry information about what is or is not the case in the world it will have to contain "propositional" expressions of some ilk (i.e., representational vehicles for expressing such information). Thus, for the purposes of this paper, a knowledge base can be defined as a set of propositional expressions in some KR language. The central insight driving the idea of an interlingua is that the content of a knowledge base B typically will be expressible as a set of sentences in a first-order language, the language of ISyCL (or some ISyCL extension) in particular, augmented (usually) by whatever distinguished predicates, constants, and function symbols are needed to represent the corresponding elements of B. A translator for B will return exactly that set; that is, it will return a first-order theory that (ideally) has precisely the information content of B.

We can make this somewhat more precise. The notion of information content, of course, gets us into issues of meaning and, in the context of logic, that means model theory. There are more sophisticated logical methods and tools for getting at the notion of content, but a good, simple picture builds on the notion of an interpretation above. Since it is assumed that propositional expressions of L correspond in some way to sentences of a first-order language, there must syntactic or structural elements of those propositional expressions corresponding to predicates, function symbols, constants, and (perhaps very limited) quantification. A translator t* for a knowledge base B into ISyCL will thus be defined in terms of some function t on the more basic syntactic elements of L, where t is defined in a manner analogous to the definition of an interpretation function π for a language L0 into a theory T1 given in the clauses above. Call such a function t a *lexical translation* of L into ISyCL. It is assumed that even if L does not contain explicit quantifiers, t nonetheless supplies some formula $t_\forall$ of ISyCL that intuitively, like $\pi_\forall$ in the definition of an interpretation, determines a domain of individuals for a model of the ISyCL translation of B.

Any worthwhile representation language will capture its intuitive semantics in a well-defined model theory. That is, more specifically, a KR language L it will provide a clear definition of a *structure* S = (S,V) for L, where S is a set (or possibly a more complex algebraic structure of some sort) and V is function that assigns meanings in S to the basic syntactic elements of L. Given a structure, one can then define what it is for a propositional expression to be *true* in such a structure in terms of the meanings of the more basic syntactic elements that make up those expressions. That is, given a structure S for L and a propositional expression e of L, one can define a metalinguistic predicate TrueIn(e,S). For a knowledge base B, one can then define TrueIn(B,S) just in case TrueIn(e,S) for all $e \in B$.

Now, let t be a lexical translation of L into ISyCL, and let M be a model of ISyCL. Given t and M, it is possible to derive a semantical structure M^t for L as follows. Let the domain of M^t be the set of elements of M that satisfy $t_\forall$. Let the meaning of a given n-place predicate (or predicate-like) expression P in M^t be the set of n-tuples of M that satisfy the corresponding ISyCL formula provided by t; similarly for those expressions that play the roles of function symbols and constants. Let Γ be a set of sentences of ISyCL, let t be a lexical translation of L into ISyCL, B a knowledge base in L, and define $t^*[B] = \{t^*(\varphi) \mid \varphi \in B\}$. Then we can say, finally, that B and Γ *have the same information content* (relative to t) just in case, for any model M of ISyCL, TrueIn(B,M^t) iff M is a model of Γ. t^* is a *translator* for B into ISyCL just in case B and $t^*[B]$ have the same information content relative to t. A translator from ISyCL to L can be defined simply as the inverse (pruned, if necessary, to ensure it is a function) of some translator t for the empty set into ISyCL. (It follows from this that the inverse of t^* is a translator from ISyCL to L, for any lexical translation t of L into ISyCL.) Such a translator is a translator from ISyCL to a knowledge base B just in case it is the inverse of a translator from B to ISyCL.

Given the notion of a translator, we can define a simple picture of model translation. The idea is that, given two knowledge bases B1 and B2, ISyCL provides a framework that subsumes them both in the sense that there are translators t_1, t_2 from both B1 and B2 into ISyCL (augmented, of course, to include constants, function symbols, and predicates as needed to express the content of B1 and B2) and translators r_1, r_2 from ISyCL to B1 and B2. Then a piece of information carried by an expression e of B1 can be translated into B2 just in case $r_2(t_1(e))$ is defined.

This page intentionally left blank.

ACRONYMS

A

AA, 308

ARPA, 279

B

BM, 221

BOM, 254

C

CIM, 305

CIMOSA, 305

COTS, 219

CYC, 304

D

DB, 303

DM, 221

E

EC, 308

EE, 308

EI, 301

F

FM, 309

I

IICE, 279

K

KBSI, 279

KIF, 279

KQML, 308

M

MCC, 301

MP, 322

N

NS, 308

O

OMT, 218

OOA/OOD, 219

OOSE, 218

OS, 260

P

PACT, 307

R

ROI, 234, 271

S

SDM, 233

SHADE, 306

SKB, 306

SM, 221

T

TOVE, 303